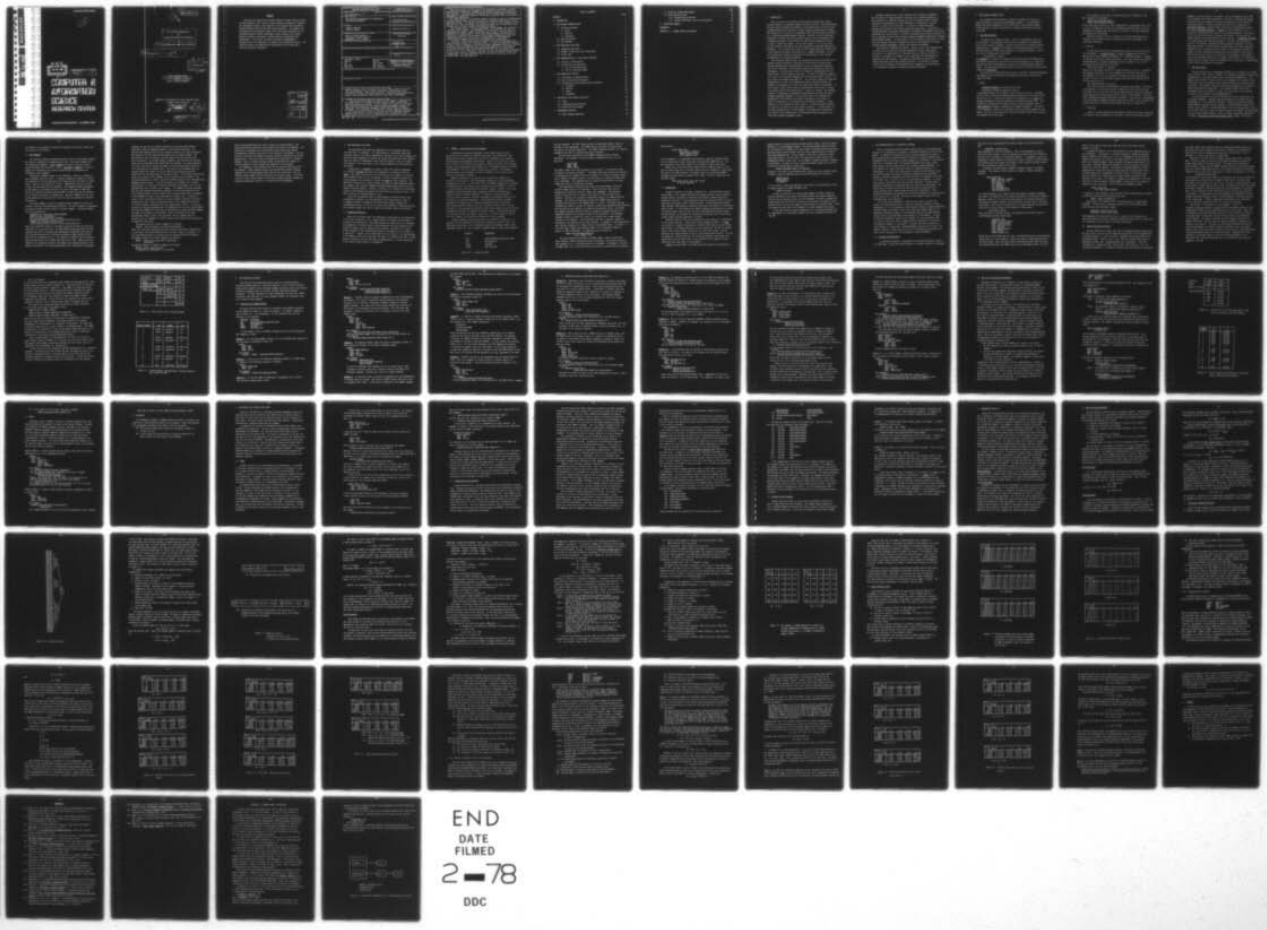
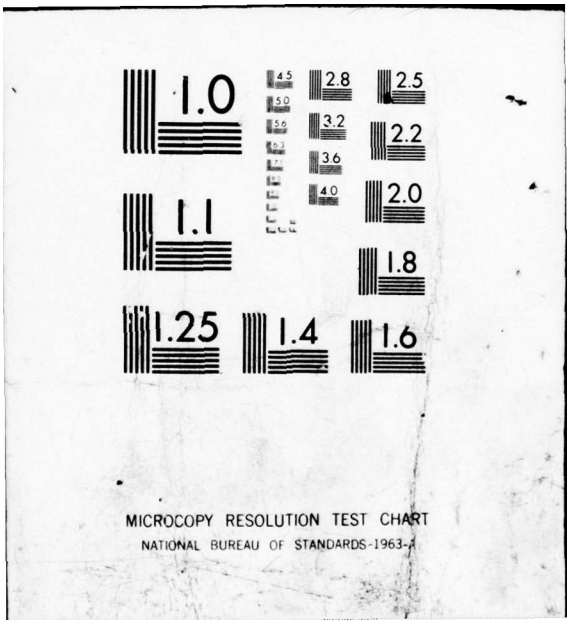
AD-A049 180 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
DBC SOFTWARE REQUIREMENT FOR SUPPORTING RELATIONAL DATABASES.(U)
NOV 77 J BANERJEE, D K HSIAO N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-77-7 NL

| OF |
AD
A049180



END
DATE
FILMED
2-78
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL REPORT SERIES

(13)

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(14) OSU-CISRC-TR-77-7

(13)

(6) DBC Software Requirement
for
Supporting Relational Databases,

by

(10) Jayanta Banerjee [redacted] David K. Hsiao

(9) Technical rept.,

(15) Work performed under
Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

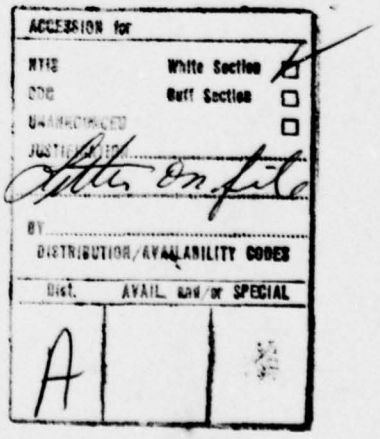
(11) Nov [redacted] 77

(12) 88 p.

407 586

PREFACE

This work was supported by contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Associate Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation



SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: OSU-CISRC-TR-77-7 ✓	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: "DBC Software Requirements for Supporting Relational Databases"	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Jayanta Banerjee, David K. Hsiao	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0573 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Washington, D. C. 20360	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 784115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: November 1977
	13. NUMBER OF PAGES: 82
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicao
ONR 102IP	ONR, Pasadena

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
Database Computer, DBC, relational data model, DBMS, System R, security, view, authorization, tuple, record, attribute, attribute-value pair, performance analysis, clustering, directory, content-addressing, associative search, query, predicate, query execution time, memory requirement.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
This is the final report of a series of work aimed at demonstrating the capabilities of a back-end database computer (DBC) in supporting known data models and systems. In the previous two reports, it was shown that existing hierarchical and network database management systems, in particular, the Information Managment System (IMS) of IBM and DMS1100 of UNIVAC, can be supported on the DBC with a vastly improved performance. In this final report, we study a relational database management system, namely System R, with a view to supporting such a system on the DBC.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

The early sections of this report are introductory in nature. A brief description of the DBC and a summary of the important aspects of System R have been presented. They have been included so that a reader without a detailed knowledge of a relational system or the DBC may follow the rest of the material without undue difficulty.

The representation of relational tuples in the DBC is quite straightforward. The data items of every tuple are converted to attribute-value pairs to form a single DBC record. Two special attribute-value pairs are also included in each DBC record in order to indicate the relation to which the corresponding tuple belongs, and to provide certain clustering information.

User transactions in the data sublanguage, called SEQUEL, are converted to a series of DBC commands. The commands are so structured that the DBC can simultaneously access a number of records, the contents of which satisfy the predicates in a SEQUEL query. Given a particular command, the DBC uses its directory to determine the portions of its secondary storage that need to be content-searched. The report further demonstrates how view mechanism, authorization, integrity assertions and triggers may be supported.

The report is concluded with an analysis of the memory requirements and query execution times in two different cases: (1) when a conventional computer system is used to implement a relational database management system and (2) when the DBC is used in conjunction with a front-end computer to do the same job. It has been observed that the mass memory requirement of the conventional system is 0.5 to 1.0 times that of the DBC, but the directory memory requirement is one or more orders of magnitude greater than that of the DBC. Under usual circumstances, the query execution time of the DBC is also faster by ten to hundred times, and sometimes more.

TABLE OF CONTENTS

1. INTRODUCTION

This is the last of a series of three reports dealing with certain software aspects of a database computer, known as the DBC. More specifically, the main theme underlying the studies is the demonstration of the fact that a database machine like the DBC is indeed capable of supporting the common data models at a cost which is considerably less than what would be incurred on a conventional general-purpose computer. Moreover, certain additional associative retrieval and access control features can be incorporated into a database management system without any significant overhead if the DBC is used to support the system. In the first report [1] of this series, the DBC software requirements have been presented for handling hierarchical databases. In the second report [2], the DBC capabilities were studied with a view towards the support of network (e.g., CODASYL) databases. We shall now conclude this series by directing our investigation on the software requirements for relational databases.

The relational model of data, as introduced by Codd [3] in 1970, is an approach towards providing a data model or view which is divorced from various implementation considerations as well as providing the database user with a high-level, set-at-a-time (rather than record-at-a-time) data sublanguage. Currently there exists no commercial implementation of any system based on the relational data model. The lack of commercial adaption is not due to any inherent inadequacy of the relational model, since it is a very simple and elegant data model [4]. The reluctance among the commercial organizations to accept and implement relational database systems can, perhaps, be attributed to the fact that other data models are available. It will require a major effort from the organizations and the users to adapt a new system such as the relational one. There are, however, two important attempts at designing and implementing experimental prototype relational database management systems. One is called System R [5] designed and implemented at the IBM San Jose Research Laboratory and the other is Ingres [6] developed at the University of California, Berkeley. In many respects, the two systems are quite similar. We shall therefore restrict our attention to only one of them, namely, System R. In other words, for the sake of specificity, we shall assume, and perhaps quite justifiably, that System R does possess most of the important features expected in a relational database management system.

Database computers are a recent addition to the family of computers. With the advent of large databases, there has been a growing awareness of the necessity of a computer architecture that is oriented towards storage, retrieval and manipulation of large quantities of information. The DBC [7,8,9] is a step in that direction. It utilizes content-addressable memories and processors with various speeds and capacities. In addition it provides powerful clustering mechanisms for performance enhancement and security mechanisms for access control. The built-in hardware data structure enables the DBC to interface directly with existing database management application programs with minimal software. In other words, it is the purpose of this report to show that the required software is minimal and that the new software can replace existing database management systems with improved performance.

This report is organized as follows. Sections 2 and 3 are introductions to the DBC and the relational database management systems, respectively. In Section 4, we demonstrate how relational data is stored in the DBC. In Section 5, we discuss how commands in the relational data sublanguage are translated into DBC queries. We propose a methodology in Section 6 to support relational views and integrity features. We conclude in Sections 7 and 8 with an analysis of DBC performance as compared to the performance of a conventional computer in managing relational databases.

2. THE DATABASE COMPUTER (DBC)

As a special-purpose computer, the DBC is intended to be used as a back-end machine to a front-end conventional computer. It is designed to handle very large databases of 10^9 to 10^{10} bytes in an efficient manner. In this section, we shall concentrate on the major architectural features of the DBC.

2.1 The DBC Data Model

Let there be two primitive sets: a set AT of "attributes" and a set VA of "values". The meaning of the two sets is assumed to be understood and is left otherwise undefined in order to allow for the broadest possible interpretation. A record R is a subset of the Cartesian product AT x VA, with the restriction that every attribute in a record is distinct. Thus, R is a set of ordered pairs of the form:

<an attribute, a value> .

The keywords of a record (or a group of records) are those attribute-value pairs which characterize the record (or the record group), i.e., those pairs that may be used to distinguish the record (or the record group) from all others. The other attribute-value pairs of a record, if any, are collectively called the record-body.

The set of all records which are stored in the DBC is called the database. The database may be partitioned into subsets called files, each with its unique file-name.

A. Query

A keyword predicate is a triple of the form:

<attribute, relational operator, value >.

A relational operator is an element of the set $\{=,\neq,<,\leq,\geq,>\}$. A keyword <A,V> is said to satisfy a keyword predicate <Ap,Op,Vp> if and only if A=Ap and V Op Vp, i.e., V and Vp are related by the operator Op. A query is a Boolean expression of keyword predicates in disjunctive normal form. Thus, a query is a disjunction of conjuncts known as query conjuncts, where a query conjunct is simply a conjunction of keyword predicates. A record in a file satisfies a query if it satisfies at least one query conjunct in the query. The set of all records in a file that satisfy a query will be called the response set of the query.

As an example of the types of queries that may be recognized by the DBC, consider the following:

([DEPT='TOY']&[SALARY<10000]) ∨
([DEPT='BOOK']&[SALARY>50000]).

If the above query refers to a file of employees of a department store, then it will be satisfied by records of the employees working either in the toy department and earning less than 10,000, or working in the book department and making more than 50,000.

Queries are used not only to retrieve a set of records among all the records in the database but also to specify protection requirements and clustering conditions.

B. Security

The DBC allows for security specifications based on the actual contents of the database. A database access or simply an access is the name of a DBC operation which transfers information to or from the database. Examples of accesses are retrieve, insert and delete. For every user of the database, the DBC maintains a database capability, which is simply a list of file sanctions whose entries are of the form:

(F,[Q1,A1],[Q2,A2],...,[Qn,An])

where F is a file name, each Qi is a query and each Ai is a set of accesses. The database capability of a user determines the records he can access. For example, for a user to be allowed to perform an access operation a on record R of file F, the following condition must hold for every (Qi,Ai) in the file sanction for F:

If (R satisfies Qi) then (a ϵ Ai).

This type of security specification is powerful and elegant. With this specification, not only can security be enforced in terms of record types or entire files, but security can also be facilitated at a much more detailed level based on the actual content of the records in the database. And since such a mechanism is directly provided in the DBC, it may easily be incorporated into any database management system supported by the DBC. A more detailed and formal discussion of the DBC security provisions will be found in [7].

C. Clustering

Based on certain prespecified information created by the user, clustering of records is done automatically by the DBC, so that records being accessed

together are stored close to one another. This is necessary since the DBC is not designed to be fully associative. The user is provided some degree of control over the placement of records by application of the concept of clustering keyword. Certain attributes of a file may be designated as clustering attributes. Keywords whose attributes are clustering attributes are termed clustering keywords. A cluster is then defined as a set of records all of which have the same set of clustering keywrods. Each record in the file will then belong to one and only cluster. The user may now impose weighted clustering conditions on the records. A clustering condition is a query formed with clustering keywords. The user, when inserting a record in the database, specifies certain clustering conditions and their associated weights. A sum-of-weight corresponding to the above clustering conditions can be calculated for any record in the database by adding the weights of those clustering conditions that are satisfied by that record. The new record (the record to be inserted) is then placed in the database close to an existing record with the largest sum-of-weight. The clustering process does not really require the inspection of the database, as demonstrated in [9], since directories are maintained.

2.2 DBC Architecture

The most natural way of addressing information in a database is in terms of the content of the records. However, the secondary storages of conventional computers have so far been limited only to location-addressability. This implies that in order to find a record in the database, the location of the record must first be determined via software techniques and auxiliary data structures. The overhead, therefore, includes the complexity of software to support auxiliary data structures. This overhead becomes particularly intolerable when the database is large, since the search of the auxiliary structure itself becomes a time-consuming process.

The DBC provides for the entire database an on-line storage which can be content-addressed. Although associative memory also provides content-addressing, it is not possible to develop a monolithic associative memory with sufficient capacity for DBC storage. By partitioning the memory into blocks, each of which is content-addressable, and by limiting access to only one of these blocks at a time, the DBC can achieve some degree of associativity and very large storage capacity. Such a processor and memory organization is termed a partitioned content-addressable memory (PCAM). The on-line

mass memory (MM) of the DBC is a PCAM. Each partition of the MM is called a minimal access unit (MAU). As an example, a 10^9 byte database will have 1,000 MAUs each of which processes and stores 10^6 bytes, which is the approximate size of a disk cylinder.

Another major component of the DBC is a processor called the database command and control processor (DBCCP). When a command from the front-end computer (the one which interfaces with the user) is sent to the DBC, the DBCCP decodes the command, determines the MAUs to be searched in order to satisfy the command, issues appropriate orders to the MM and transfers data to/from the front-end computer.

Since a large database will contain many MAUs and since only one MAU can be accessed at a time, it is not practical to search all the MAUs for each search order. Hence, directory entries are made for certain keywords. These keywords are called Type-D keywords or directory keywords. A directory entry consists of a Type-D keyword and the numbers of the MAUs in which records containing this keyword appear. Any query conjunct is expected to have at least one predicate consisting of a directory keyword. Otherwise, an exhaustive search of the MM will be necessary to satisfy the query. In addition, the clustering keywords and security keywords are treated as instances of Type-D keywords.

The collection of all the directory entries is also stored in a PCAM with a capacity and processing speed that is different from the mass memory PCAM. This PCAM is known as the structure memory (SM). Typically directories are of the order of 1% to 10% of the database. Therefore, the SM has a capacity of 10^7 to 10^9 bytes. It is estimated that a query conjunct will seldom have more than 20 predicates; and a single MAU access will normally satisfy a query. Therefore, the access speed of the SM is about 1 millisecond which is about 20 times faster than the time required to access an MAU. Thus, in the time required to process a query in the SM, another query may be satisfied by accessing an MAU. The relationship of SM, MM and DBCCP is depicted in Figure 2.1.

The processors associated with the MM have the capability of returning a group of records (satisfying a query) in a sorted order, say, sorted by a given attribute. They can also carry out certain set functions. In particular, they can take a group of records and determine the minimum, maximum, sum and average of the various values of a given attribute considering all the records in the group. The number of records satisfying a query can also be counted by hardware. Furthermore, any specific combination

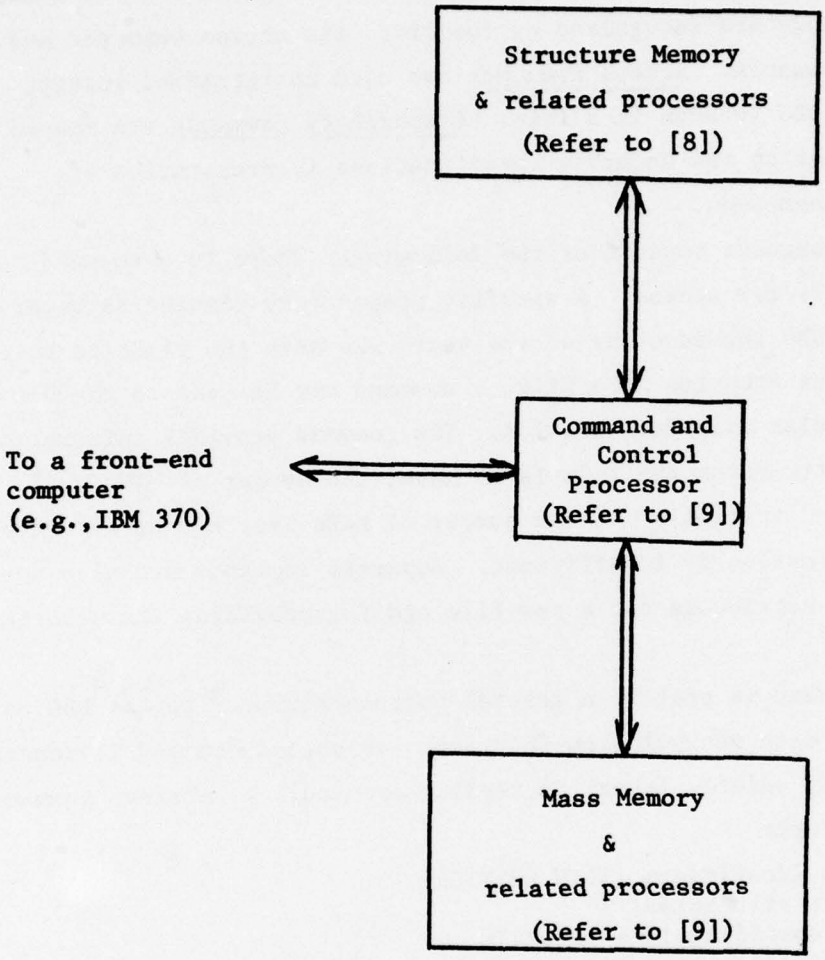


Figure 2.1. Basic architecture of the DBC

of fields of a record may be returned (on request) to the user, rather than the record in its entirety.

2.3 DBC Commands

The front-end computer communicates with the DBC by issuing DBC commands. Two types of commands are recognized by the DBC: the access commands and the preparatory commands. Access commands are used to retrieve, insert, delete and update DBC records in a file. Preparatory commands are issued to manage file information and security specifications in preparation of subsequent access commands.

Preparatory commnads consist of the following. There is a command to open a database file for access. A specific preparatory command is reserved for informing the DBC the identity of the users who have the right to create files. Prior to the creation of a file, a command may be sent to the DBC to open the particular file for creation. The command provides information on the number of attributes the file is to have, the number of MAUs that need to be allocated initially and the number of MAUs that may be allocated if the initial allocation is insufficient. Separate commands are also used for specifying the attributes for a new file and for providing its security descriptors.

An access command is sent by a general-purpose computer to the DBC to perform a specific data manipulation function. An access command is recognized as being a retrieve, delete, insert or replace command. A retrieve command has the following form:

```
RETRIEVE:<file identifier> [WITH POINTER]
[[SORT BY <sort attribute>] /
[set function specification> [ONLY]]]
[<field specification list-1>] <record specification-1>
[CONNECT ON <attribute-1,attribute-2>
[<field specification list-2>] <record specification-2>]
```

where file identifier refers to the name of the file on which the retrieval operation is to be carried out: the WITH POINTER clause specifies that the response data must be accompanied by implemntation-dependent pointers; the sort attribute specifies the attribute according to whose values the DBC must sort the response data (i.e., the records or fields); the set function specification may be one of COUNT, AVG, MAX, MIN or SUM. COUNT returns the number of data elements retrieved, AVG computes the average value(s) of the field(s) specified in the field specification list, MAX returns the maximum

value(s) for each of the field(s) specified in the field specification, while MIN returns the minimum value(s) of the field(s) specified. The function SUM computes the sum of the values for each of the fields specified. The functions AVG, MIN, MAX and SUM operate only on numeric fields. The ONLY clause is used to indicate that the value of the set function need only be returned. If the ONLY clause is omitted, then both the set function value and the individual field values will be returned. The field specification list-1 and list-2 specify the names of the attributes whose values are to be retrieved as the response data. The values in the response data can be made unique with respect to any one field by associating the prefix UNIQUE with the attribute name of the desired field. If the field specification lists are not given in the command, entire records will be retrieved. The record specification is either a query (as defined earlier) or an implementation-dependent record pointer. A record retrieved in accordance with a record specification or a collection of fields retrieved in accordance with a field specification list is called a data element. The CONNECT ON clause specifies that the data elements retrieved in response to record specification-1 and record specification-2 must be joined on the attributes specified as arguments of the CONNECT ON clause. Here join means an equality join. Attribute-1 refers to the connecting attribute in the data elements defined by the field specification list-1, while attribute-2 refers to the connecting attribute in the data elements defined by the field specification list-2. In the case when the set function specification, the UNIQUE option in the set specification lists, and the CONNECT ON clause are all specified in a single retrieve command, the order of precedence is as follows: First, the fields in the field specification lists (1 and 2) are extracted, then the UNIQUE option is executed, next the CONNECT ON clause is effected and finally the set function is applied to the result.

The general form of a delete command is as follws:

DELETE: <file identifier> [<record specification>]

where file identifier identifies the file on which the deletion operation is to take place, and record specification is either a query or a pointer. If the record specification is omitted, then the entire file will be deleted. The general format of an insert command is as follows:

INSERT: <record to be inserted> [<clustering conditions>]

The general format of a replace command is as follows:

REPLACE: <record specification>
<keywords for replacement> / <new record >

where record specification is either a query or a record pointer. The record specification describes the record(s) which need to be modified. The keywords for replacement are attribute-value pairs which will replace corresponding attribute-value pairs in the records to be modified. The new record is one which will replace the entire record(s) defined by the record specification. Only one of the two options may be specified in a replace command. That is, either existing records are modified with respect to certain keywords or entire records are replaced by a new record.

The commands we have discussed thus far are executed directly by the DBC hardware. These commands deviate from conventional machine language commands in the following major ways: they are very high-level, they use variable length formats and they provide set-at-a-time access. In Sections 5 and 6, we shall have occasion to use these extremely powerful commands in executing transactions written in a relational data sublanguage.

3. THE RELATIONAL DATA MODEL

The relational data model can simplify both the conceptual view and the user view of a database. All relationships or connections among data items are shown in the form of mathematical relations over a set of domains [3]. The data items themselves are simply the values associated with these domains. Thus, the number of primitives in the relational model is only one, namely, the relation. This contributes to the overall simplicity of the relational data model.

Conceptually, a relation is a table in which each column corresponds to a distinct attribute and each row corresponds to a distinct entity or tuple. Each tuple is distinct in the sense that no two tuples in a relation have identical values for all attributes. The set of possible values that can be assumed by an attribute is called the domain of that attribute. Two different attributes of a single relation can have the same underlying domain. For example, the attributes QUANTITY and RUNS-SCORED assume values from the domain of natural numbers. Finally, a relation R is a subset of the Cartesian product of the domains associated with the relation's attributes A1,A2,...An. Such a relation is denoted R(A1,A2,...,An).

There are two important ways in which database relations differ from mathematical relations: (1) The ordering of the values within a tuple of a database relation is immaterial if the attribute names accompany the corresponding values; (2) The set of tuples that comprises a database relation will normally change over time as tuples are inserted, deleted or modified.

3.1 Normalized Relations

While the relational model can inherently be applicable to the formulation of any type of relation, it is a common practice to subject the relations to a process of normalization. The primary reason for doing this is to eliminate the possibility of certain types of inconsistencies that may otherwise arise during the update (such as, modification, deletion, insertion) of a tuple. A fine treatment of the normalization process is presented in [4] as well as in [10,11,12]. However, since normalization is not central to an understanding of the manner in which the relational data model is handled by the DBC, we shall restrict ourselves, only for the sake of completeness, to a brief description of the various normal forms in the appendix.

3.2 SEQUEL: A Relational Data Sublanguage

A comprehensive database management system (DBMS) should include provisions such as simple but flexible user views, data definition, data manipulation and query capabilities, as well as convenient access support, system recovery and integrity enforcement. System R [5] is one such system that is based on the relational data model. System R provides user interface through a data sublanguage called SEQUEL [13]. An improved version of SEQUEL, called SEQUEL 2 [14], is currently being used in Sytem R. The language features used in this document are those of SEQUEL 2, even though we shall refer to them simply as SEQUEL.

SEQUEL is designed to be used both as a stand-alone language for interactive users and as a data sublanguage embedded in a host programming language such as PL/I. In the latter case, the SEQUEL statements in a program are identified by a precompiler which replaces them with valid PL/I calls to a run-time module which performs the desired function. SEQUEL, as its name suggests (Structured English Query Language), provides extensive query facilities based on English keywords. In addition, a data manipulation facility permits insertion, deletion and update of individual tuples or sets of tuples in a relational database. A data definition facility permits definition of relations and of various alternative views of relations. A data control facility permits each user to authorize other users to access his data; it also provides for assertions about data integrity, and for stored transactions that may be triggered by various events. The language operates on relations in first (or higher) normal form. We shall briefly illustrate the use of SEQUEL by writing some transactions in this language.

A sample database, extracted from [14], is depicted in Figure 3.1. It consists of four normalized relations (not necessarily in 2NF or 3NF). The EMP relation describes a set of employees, giving the employee number, name, department number, job title, manager's employee number, salary and commission

Relation	Attributes
EMP	EMPNO,NAME,DNO,JOB,MGR,SAL,COMM
DEPT	DNO,DNAME,LOC
USAGE	DNO,PART
SUPPLY	SUPPLIER,PART

Figure 3.1. A Sample Database

for each employee. The DEPT relation gives the department number, name and location of each department. The USAGE relation describes the parts which are used by the various departments. The supply relation describes the supplier companies from which the various parts may be obtained.

The most basic operations of the SEQUEL language involve the query facilities. For example, to find the names of employees in Dept. 50, one may write

```
SELECT NAME
FROM   EMP
WHERE  DNO=50
```

The SELECT clause lists the attributes to be returned. If the entire tuple is desired, one may write SELECT *. The WHERE clause may contain any collection of predicates which compare values of attributes of a tuple to constant values (e.g., DNO=50) or compare values of two attributes of a tuple with each other (e.g., SAL<COMM). The predicates may be connected by AND and OR, and parentheses may be used to establish precedence.

Data manipulation facilities are those facilities whereby a user may directly change values in the database. These facilities fall into the categories of insertion, deletion, update and assignment. The insertion facility allows the user to insert a new tuple or a set of tuples into a relation. Deletion is a process of specifying tuples to be removed from the database. The tuples are specified by means of a WHERE clause which is syntactically identical to the WHERE clause of a query. The update features of SEQUEL are similar to those for deletion, except that additional specifications must be given for the updates to be made on the selected tuples. New values for updated attributes may be stated as constants, as nested queries or as expressions based on the original values of the attributes. An assignment statement allows the result of a query to be copied into a newly-created relation in the database. The new relation may then be queried, updated or processed in the same way as any other relation. An example of data manipulation that involves insertion of a single tuple is illustrated below. To insert a new tuple named 'JONES' with employee number 535 in department number 51, having no other attributes, a transaction may be written as

```
INSERT INTO EMP(EMPNO,NAME,DNO):
        <535, 'JONES', 51>
```

The data definition facilities of SEQUEL enable users to create and drop relations, define alternative views of relations, and specify the access aids (indexes, etc.) to be maintained on the database. For example, to create the DEPT table (i.e., relation) during the process of constructing the database,

one may write

```
CREATE TABLE DEPT
        (DNO (CHAR(2), NONULL),
         DNAME (CHAR(12) VAR),
         LOC (CHAR(20) VAR))
```

In this example, it is indicated to the system to create a relation (which is to be physically stored) with three attributes DNO, DNAME and LOC. The DNO attribute of any tuple of DEPT is not allowed to take a null value.

SEQUEL data control facilities enable users to control access to their data by other users, and to exercise control over the integrity of data values. The owner of the EMP relation, for example, may use the following SEQUEL statement to grant Smith and Anderson the right to read or update the JOB and DNO columns:

```
GRANT READ, UPDATE (JOB, DNO) ON EMP
      TO SMITH, ANDERSON
```

3.3 Access Aids

System R consists of two major components [5]. The Relational Storage System (RSS) is the storage subsystem that manages devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, backout and recovery. It also maintains indexes on selected fields of stored relations, and pointer chains across relations. The Relational Data System (RDS) provides authorization, integrity enforcement and support for alternative views of data. It also supports the SEQUEL language and maintains the catalogs of external names, since the RSS uses only system-generated internal names. The RDS contains an optimizer for choosing an appropriate path for any given request among the paths supported by the RSS.

System R relies on the user to specify the access paths to be maintained on the stored relations. Access paths include images and links. An image in the RSS is a logical reordering of a relation with respect to one or more sort fields. It provides associative access capability. The RDS can rapidly fetch a tuple from an image by keying on the sort field values. The RSS maintains each image through the use of a multi-page index structure. At most one image on a relation may have the clustering property, which means that tuples which are near each other in the ordering of that image are stored physically near each other in the database.

Links are access paths in the RSS which link tuples of one relation to

related tuples of another relation through pointer chains. Links are always employed in a value-dependent manner: the user may specify that each tuple of a relation may be linked to all those tuples of another relation that have matching values in some field(s) and that these tuples should be ordered in some value-dependent way. Like an image, a link may be declared to have a clustering property, in which case, the tuples will be kept close to the neighboring tuples in that link. As an example of images and links, consider the following. In order to create a clustering image, named IM, on the SAL attribute of the EMP table, we may write

```
CREATE CLUSTERING IMAGE IM ON EMP(SAL).
```

Similarly, to create a non-clustering link, called LK, which connects tuples of DEPT to tuples of EMP that match on the DNO attribute, we may write

```
CREATE LINK LK
FROM DEPT(DNO)
TO EMP(DNO).
```

If we are also to order the employees on the link by, say JOB and SAL, then we simply expand this SEQUEL statement with

```
ORDER BY JOB,SAL.
```

It must be noted that the access paths (images or links) contain no logical information that cannot be derived from the data values themselves. The user has no explicit control over the placement of tuples in images and links (except for the ability to declare the structure of an image or link). Neither can the user use the image or link directly for accessing data. Links and images are used only by the optimizer to choose optimal access paths.

In the next section we shall demonstrate how a relational database may be transformed to an equivalent database that can be supported directly by the DBC.

4. DBC REPRESENTATION OF A RELATIONAL DATABASE

Given a conventional general-purpose computer with location-addressable storage, it is convenient to represent a relational database in the following manner. A template is maintained for each relation indicating the name of each attribute and its relative position within the relation (besides other information such as the type of values assumed by an attribute). The relative position is a necessary part of attribute information since each stored tuple carries only the values and not the attribute names themselves. All the elements of a single tuple are stored in a physical sequence, i.e., contiguously. In order that tuples may be retrieved without having to scan the entire database, two kinds of auxiliary information are maintained. A separate index may be maintained for any attribute of a relation in the form of an m-ary tree. Any leaf of such a tree provides the address to a stored tuple having a particular value for the given attribute. Another type of auxiliary information is maintained directly within the stored tuples. The tuples that are related by the fact that they have the same value for a given attribute may be linked by means of pointers (addresses). These auxiliary information are managed by the system according to the specifications of the creator of a relation. The user of the database need not have any knowledge about their existence.

In the DBC, all information is stored in the form of records that consist of attribute-value pairs. Since the secondary storage is content-addressable, it is not necessary to have separate indexes or address-dependent pointers within the records. Since the on-line mass memory (MM) is not a monolithic associative memory, it is desirable that only one MAU (minimal access unit) be accessed during the execution of a DBC command. In order to avoid an exhaustive search of all the MAUs, the structure memory (SM) of the DBC is utilized. With a proper choice of keywords to be entered in the structure emmory, it is possible to restrict the search required for most queries to a single MAU. Address-dependent pointers are, in fact, not needed. However, a small directory in the SM, automatically managed by the DBC, is still essential to the achievement of high performance.

4.1 Creation of DBC Records

A relational database is represented in the DBC by creating a record for each tuple. Since each tuple belongs to a certain relation, we represent

this fact in the corresponding DBC record by means of the attribute-value pair

<RELATION, relation-name>,

where RELATION is a built-in attribute. With the incorporation of such a tuple into every record, it will be possible to retrieve any record based on the fact that it belongs to a particular relation. In response to any query involving a certain relation, the DBC will be able to conduct a content-search and retrieve all those records that belong to that relation and that satisfy the other conditions required by the query.

A relation (or table) may be defined in SEQUEL by means of a CREATE statement. For example, the EMP relation of Figure 3.1 may be defined as follows:

```
CREATE TABLE EMP
     (EMPNO (INTEGER , NONULL),
      NAME (CHAR (12) VAR),
      DNO (CHAR(2)),
      JOB (CHAR(6)),
      MGR (INTEGER),
      SAL (DECIMAL(8,2)),
      COMM (DECIMAL(8,2))).
```

In a DBC environment (i.e., where a front-end computer accesses a database via a DBC), the definition of the table is retained by the front-end computer. Whenever a tuple is to be stored in the database, the software interface in the front-end computer creates a DBC record which consists only of attribute-value pairs. Such a pair is created for the relation name and one attribute-value pair for every column of the relation, as shown below:

<column-name, value>.

Thus, any tuple of the EMP relation is represented in the DBC by means of the following attribute-value pairs:

```
<RELATION, EMP>
<EMPNO, employee-number>
<NAME, employee-name>
<DNO, department-code>
<JOB, job-code>
<MGR, manager-number>
<SAL, salary>
<COMM, commission>.
```

If any one of the columns does not have a corresponding value in any particular tuple, then it is not necessary to create (or store) an attribute-value pair for that column. Thus, every DBC record representing an EMP tuple will have an attribute-value pair for EMPNO (since this column always takes a non-null

value), but it may not have such a pair for DNO (if the value for DNO happens to be null).

A significant feature of the data definition of the relational model is its flexibility. Occasionally, it becomes necessary to expand an existing table by adding a new column to it, e.g., to accommodate a new application. SEQUEL allows columns to be added to the right side of an existing table by means of an EXPAND statement, which specifies the name and data type of the new column. Existing tuples are considered to have null values in the new column until they are updated. This feature is easily handled in the DBC representation, since only records representing the new tuples will have an attribute-value pair corresponding to the new column. Old records, until they are updated by a transaction, will remain unaltered inspite of the expansion of the table. For example, a change in the definition of the DEPT table may be made by adding a column (NEMPS) to keep track of the number of employees in each department. This is done by the SEQUEL statement

```
EXPAND TABLE DEPT
     ADD FIELD NEMPS(INTEGER).
```

In response to this statement, only those DBC records representing new tuples of DEPT will have the extra attribute-value pair

<NEMPS, number-of-employees>.

Notice that it is possible that a relation may have a column called RELATION. In that case, it seems that there will be two attribute-value pairs with the same attribute, namely,

<RELATION, relation-name> and
<RELATION, value-of-the-column>.

This ambiguity will never actually occur. Since all the attributes are coded and because the built-in attribute RELATION has been given a unique code, it is different from the codes of all other attributes.

4.2 Access Aids and Clustering

System R makes use of images and links to determine optimal access paths. The DBC, however, has no use for images and links that are not designated for clustering purposes. It does not need to implement these access aids in the way System R does. This is due to the fact that the DBC uses content-addressable memory, thus eliminating the need of pointers. The only information on images and links that the DBC will make use of is the clustering information. It should be obvious that the intention in specifying

clustering images and clustering links is to physically gather together all data that will be most frequently retrieved in response to a single query. Since the DBC does not simultaneously access two or more MAUs it will be desirable to cluster such data in a single MAU.

Clustering of the DBC records, however, always starts with a relation name. We first attempt to store in as few MAUs as possible all those DBC records that belong to the same relation. The reason for clustering by relation name is simply that all SEQUEL queries involve one or more relations. Thus, the relation names being known, it will always be possible to form DBC commands with the query field consisting of at least one predicate of the form

(RELATION=relation-name).

In this manner we will almost always be ensured that the DBC will satisfy any given query by accessing at most a number of MAUs that is no greater than the number of MAUs required to store all the records in a given relation. The actual number of MAUs accessed will, in fact, be usually less than this number. We observe, therefore, that DBC records are clustered primarily by relation name. This clustering is done while inserting each record, say, belonging to relation r, by indicating in the insert command that (RELATION=r) is the primary clustering condition (which is called mandatory clustering condition in DBC terminology). Further clustering is done based on the information on the access aids (images and links), as we shall now discuss.

For records to be inserted in the database, there is a secondary clustering condition (called optional clustering condition in DBC terminology) derived from the specification of the access aids. There are no secondary clustering conditions for records of any relation for which no clustering image or clustering link has been specified in the definition of the database. We assume, in keeping with the language definition of SEQUEL, that there is no more than one clustering image or clustering link defined on any relation. If there are, then we may arbitrarily pick one of them (in determining a secondary clustering condition) since it normally does not pay to have more than two levels of clustering in a PCAM (partitioned content-addressable memory) with large partitions. Assuming that we have decided on a particular image or link for clustering purposes, let us illustrate what clustering conditions are to be provided by the front-end computer to the DBC.

A. Use of Clustering Links

Considering the general definition of a clustering link, namely,

```
CREATE CLUSTERING LINK link-name
FROM   relation-1(attribute-list-1)
TO     relation-2(attribute-list-2)
ORDER  BY attribute-list-3
```

we are to link every record of relation-1 to one or more records of relation-2 such that attribute-list-1 and attribute-list-2 have matching values. Since records of two different relations cannot be clustered (because the primary clustering condition is based on a single relation name),we only cluster all those records of relation-2 that belong to the same link. Thus, usually a maximum of two accesses will be necessary to access all records that logically belong to the same link: one access for the records belonging to relation-1 and another access for the records belonging to relation-2. We cluster relation-2, therefore, by attribute-list-2; and this is done as follows: Assume that the tuples of relation-2 together occupy upto N MAUs (in case no knowledge is available as to the tuple size and number of tuples in a relation, assume N to be a large number, say 200, which is likely to be large enough to accommodate any relation). In any tuple (of relation-2) to be stored in the database, add an extra attribute-value pair

<CLUSTER, hash number>,

where hash number, between 1 and N, is obtained by hashing the values of attribute-list-2. The secondary clustering condition for this tuple, then, is (CLUSTER = hash number). The primary clustering condition, we may recall, is simply (RELATION = relation-2).

B. Use of Clustering Images

The implementation of the clustering properties of an image is only a little more complex. The added complexity arises due to the fact that an image is always ordered by one or more attributes. Let us illustrate the implementation process by considering the general definition of a clustering image:

```
CREATE CLUSTERING IMAGE image-name
ON relation-1 (attribute-list)
```

Assume that the attribute-list consists of the attributes A1,A2,...,An. Then the image is to be ordered logically (as well as physically, since it is

a clustering image) first by A1, secondly by A2, etc.

If the possible values of attribute A1 have a reasonably wide range, then it is clear that clustering by A2,...,An (after clustering by A1) will give no added advantage. For example, if the values of A1 were to range uniformly between 1 and 1000 and the number of MAUs occupied by relation-1 is 20, then we may be able to cluster in such a way that records with A1-values ranging between 1 and 50 are stored in the first of the 20 MAUs, records with A1-values ranging between 51 and 100 are stored in the second, etc. Further clustering by attribute A2 serves no purpose because in any search query with equality predicates involving attributes A1 and A2, the value of A1 alone (together with relation name) is sufficient to determine the MAU that contains all the records satisfying the query.

On the other hand, if the possible values of attribute A1 have a very small range (e.g., the attribute SEX may take values either male or female), then it is advantageous to use attribute A2 for creating finer (smaller) clusters that may be accommodated in a single MAU. Similarly, if each of A1 and A2 has a small range of permissible values, then attribute A3 may be used to create finer clusters.For clustering purposes, therefore, we use attribute A1,...,Ai such that each of A1,...,A(i-1) assume a small range of values and either i=n or Ai assumes a large range of values.

To determine the cluster number of any tuple, the front-end computer makes use of a table for each relation. There are as many columns in a table as there are attributes chosen for clustering (as discussed in the last paragraph). The entries in each column have the following meaning:

Entry 1: Name of the clustering attribute.

Entry 2: Range of values of the attribute;
it is either <u>large</u> or <u>small</u> .

Entry 3: Integer representing the following:

If Entry 2 = small,
then Entry 3 = number of values in the range.

If Entry 2 = large,
then Entry 3 = number of partitions
made of the range.

Entries 4,5,...: Values of the attribute's range,
or maximum values for the partitions of the range.

As an example, consider a POPULATION relation, some of whose attributes are

SEX: male or female

STATUS: employed, ineligible or unemployed

AGE: any integer.

If this relation is to be clustered by SEX, STATUS and AGE respectively, then we may create a table as shown in Figure 4.1. There are 2*3*5 = 30 clusters for this relation as shown in Figure 4.2. The number of partitions chosen for the attribute AGE is 5. This choice should not be arbitrary but should be based on the fact that the total number of clusters required is equal to or somewhat larger than (say, double or triple) the number of MAUs occupied by the relation. In case the size of a relation (in terms of MAUs required) is unknown, then a suitably large number is assumed, say 200.

The creator of a clustering image may provide the range information on the various attributes by statements such as

RANGE OF SEX IS (MALE, FEMALE)

RANGE OF AGE IS INTEGER (SMALLEST=0,LARGEST=80)

RANGE OF NAME IS ALPHA (SMALLEST=' ',LARGEST='ZZZ')

where, in the case of integer, floating-point or alphanumeric attributes, the usual range is also specified. Any range declared to be integer, alphanumeric or floating-point is considered a large range. Any range that is actually listed out (e.g., male, female) is considered small. This RANGE specification is currently not a part of the SEQUEL CREATE statement but can easily be incorporated in the language definition in order to facilitate clustering on a partitioned content-addressable memory.

Once a cluster number C is determined for any record of relation R, then a special keyword <CLUSTER, C> is included as part of the stored record. The primary and secondary clustering conditions for this tuple, then, are (RELATION = R) and (CLUSTER = C), respectively.

In this section, we have considered the DBC representation of a relational database. Clustering of the DBC records has been considered as part of the over-all clustering problem; it has been shown how an extra keyword with attribute CLUSTER is created and stored in each record. In the next section we shall illustrate how SEQUEL queries are transformed into DBC commands .

Attribute	SEX	STATUS	AGE
Range	Small	Small	Large
Number of values or value partitions	2	3	5
Values or value partitions	female	employed	20
	male	ineligible	40
		unemployed	60
			80
			Infinity

Figure 4.1. Table created for a clustering image

Cluster Number	SEX	STATUS	AGE
1	female	employed	<20
2	female	employed	<40
⋮	⋮	⋮	⋮
5	female	employed	<infinity
6	female	ineligible	<20
⋮	⋮	⋮	⋮
10	female	ineligible	<infinity
11	female	unemployed	<20
⋮	⋮	⋮	⋮
15	female	unemployed	<infinity
16	male	employed	<20
⋮	⋮	⋮	⋮
30	male	unemployed	<infinity

Figure 4.2. Cluster numbers corresponding to various values of SEX, STATUS and AGE

5. THE TRANSLATION PROCESS

The DBC,as we have indicated earlier, acts as a back-end machine executing commands given by a front-end computer. Let us call the software package,which resides in the front-end computer and which creates and handles a relational database stored in the DBC, the RDBI (Relational Database Interface). The RDBI intercepts the data sublanguage (in this case, SEQUEL) statements, which are part of a host language program, and translates them into a series of DBC commands.

5.1 Translation of SEQUEL Queries

We shall illustrate the process of translation from SEQUEL statements to DBC commands by means of a series of examples. Our sample database is that of Figure 3.1. For convenience, we will reproduce here the four relations in the database:

Relation	Attributes
EMP	EMPNO,NAME,DNO,JOB,MGR,SAL,COMM
DEPT	DNO,DNAME,LOC
USAGE	DNO,PART
SUPPLY	SUPPLIER,PART

In each example, first the SEQUEL statement and then the corresponding DBC commands are shown.

Example 1: The following SEQUEL statement and its equivalent DBC command will find the names of employees in Dept. 50.

```
SEQUEL:
   SELECT NAME
   FROM   EMP
   WHERE  DNO=50

DBC Command:
   RETRIEVE:  (NAME)  ((RELATION='EMP')&(DNO=50))
```

Example 2: A list of all the different department numbers in the EMP table is created by the following statement or commands.

```
SEQUEL:
   SELECT UNIQUE DNO
   FROM   EMP

DBC Command:
   RETRIEVE:  (UNIQUE DNO)(RELATION='EMP')
```

Example 3: To list the names of employees in departments 25, 47 and 53. the following statement may be used.

```
SEQUEL:
   SELECT NAME
   FROM   EMP
   WHERE  DNO IN (25,47,53)

DBC Command:
   RETRIEVE:  (NAME)(((RELATION='EMP')&(DNO=25))
                     ∨((RELATION='EMP')&(DNO=47))
                     ∨((RELATION='EMP')&(DNO=53)))
```

Example 4: Consider listing the names of employees who work for departments in Evanston. This type of transaction requires access to two different relations and is, therefore, expressed in SEQUEL by means of a nested SELECT statement. The inner part of the nesting returns the collection of DNO values of the departments located in Evanston. The outer part then proceeds as though it were given a set of constants in lieu of the inner SELECT clause.

```
SEQUEL:
   SELECT NAME
   FROM   EMP
   WHERE  DNO IN
          SELECT DNO
          FROM   DEPT
          WHERE  LOC='EVANSTON'

DBC Commands:
   a.  RETRIEVE:(DNO)((RELATION='DEPT')&(LOC='EVANSTON'))
```

For each department number 'di' retrieved by (a), the RDBI issues the DBC command:

```
   b.  RETRIEVE:(NAME)((RELATION='EMP')&(DNO='di'))
```

Example 5: The employee number, name, and salary of employees in Dept. 50 may be listed as follows, in the order of employee number.

```
SEQUEL:
   SELECT EMPNO,NAME,SAL
   FROM   EMP
   WHERE  DNO=50
   ORDER  BY EMPNO

DBC Command:
   RETRIEVE: (EMPNO,NAME,SAL)
             ((RELATION='EMP')&(DNO=50))
              SORT BY EMPNO
```

In case the ordering of the response set is to be done also by some secondary attributes, then such a sorting is done by the RDBI in the front-end computer. The DBC only sorts by a single attribute.

Example 6: An important class of queries is exemplified in the determination of average salary of clerks. The built-in SEQUEL function AVG can be used to accomplish this result. Other built-in functions in the SEQUEL language

are SUM, COUNT, MAX and MIN. These functions are indeed part of the hardware DBC features.

```
SEQUEL:
   SELECT AVG(SAL)
   FROM   EMP
   WHERE  JOB='CLERK'

DBC Command:
   RETRIEVE:(AVG(SAL))((RELATION='EMP')&(JOB='CLERK'))
```

Example 7: The following statement determines the count of all the different jobs held by employees in Dept. 50.

```
SEQUEL:
   SELECT COUNT(UNIQUE JOB)
   FROM   EMP
   WHERE  DNO=50

DBC Command:
   RETRIEVE: [COUNT ONLY](UNIQUE JOB)
             ((RELATION='EMP')&(DNO=50))
```

Example 8: In addition to simple attributes and built-in functions, SEQUEL allows a user to construct arithmetic expressions in the SELECT clause. All the following are valid SEQUEL expressions:

```
AVG(SAL)/52

AVG(SAL)+AVG(COMM)

MIN(SAL+COMM)
```

Since the DBC does not have any arithmetic capabilities, all arithmetic operations are done by the RDBI in the front-end computer. For example, to execute the SEQUEL statement for the first expression, the RDBI sends one command to the DBC to retrieve AVG(SAL). It then divides the resulting number by 52. For the second expression, two DBC commands are required to retrieve two numbers, which are then added. For the third expression, a single DBC command is required to retrieve the SAL and COMM fields of each EMP record; the addition and MIN operations are then performed by the RDBI.

Example 9: Consider listing all the departments and the average salary of each. This is an example of a query in which a relation needs to be partitioned into groups. A built-in function can then be applied to each group.

```
SEQUEL:
   SELECT DNO,AVG(SAL)
   FROM   EMP
   GROUP  BY DNO

DBC Commands:
   a.  RETRIEVE:(UNIQUE DNO)(RELATION='EMP')
For each department number 'di' retrieved by (a), the RDBI issues a command:
```

b. RETRIEVE:(AVG(SAL))((RELATION='EMP')&(DNO='di'))

Example 10: Sometimes it may be desired to partition a relation into groups and then to apply a predicate or a set of predicates which chooses only some of the groups and disqualifies others. These group-qualifying predicates are placed in a special HAVING clause. A predicate in a HAVING clause may compare an aggregate property (e.g., AVG(SAL)) of a group to a constant or to another aggregate property of the same group. The following SEQUEL statement may be used to list all those departments in which the average employee salary is less than 10,000.

```
SEQUEL:
   SELECT DNO
   FROM   EMP
   GROUP  BY DNO
   HAVING AVG(SAL)<10000
```

DBC Commands:

a. RETRIEVE: (UNIQUE DNO)(RELATION='EMP')

For each department number 'di' retrieved by (a), the RDBI issues a command:

b. RETRIEVE:(AVG(SAL))(RELATION='EMP')&(DNO='di'))

Since the DBC does not make comparisons on aggregate properties, the final selection of DNO based on (AVG(SAL)<10000) is done by software (i.e., by the RDBI) in the front-end computer.

Example 11: When a query has both a WHERE clause and a HAVING clause, then the WHERE clause has precedence since it is applied to qualifying tuples, in contrast to the HAVING clause which is applied to groups of tuples. Use is made of both these clauses in listing the departments which employ more than ten clerks.

```
SEQUEL:
   SELECT DNO
   FROM   EMP
   WHERE  JOB='CLERK'
   GROUP  BY DNO
   HAVING COUNT(*)>10
```

The notation COUNT(*) denotes the count of tuples in a group.

DBC Commands:

a. RETRIEVE:(UNIQUE DNO)(RELATION='EMP')

For each department number 'di' retrieved by (a), the following command is issued:

```
b. RETRIEVE:[COUNT ONLY]
            ((RELATION='EMP')&(DNO='di')&(JOB='CLERK'))
```

The RDBI now returns the name of only those departments for which a count of greater than 10 is retrieved by (b).

Example 12: Set comparison operators like =, ≠, [IS] [NOT] IN, CONTAINS and DOES NOT CONTAIN are allowed in a HAVING clause as illustrated by this example, which lists the departments which have employees with every possible job title.

```
SEQUEL:
   SELECT DNO
   FROM   EMP
   GROUP  BY DNO
   HAVING SET(JOB)=
          SELECT JOB
          FROM   EMP
```

```
DBC Commands:
   a.  RETRIEVE:(UNIQUE DNO)(RELATION='EMP')
   b.  RETRIEVE:(UNIQUE JOB)(RELATION='EMP') SORT BY JOB
For every department number 'di' retrieved by (a), issue the command:
   c.  RETRIEVE:(UNIQUE JOB)
                ((RELATION='EMP')&(DNO='di')) SORT BY JOB
```

For each department, the comparison of each of the sets in (c) to the set in (b) is done by software (i.e., by the RDBI).

Example 13: The set theoretic operators INTERSECT, UNION and MINUS are also available in SEQUEL. Consider, for example, the listing of all the departments which have no employees.

```
SEQUEL:
   SELECT DNO
   FROM   DEPT
   MINUS
   SELECT DNO
   FROM   EMP
```

```
DBC Commands:
   a.  RETRIEVE:(UNIQUE DNO)(RELATION='DEPT')
   b.  RETRIEVE:(UNIQUE DNO)(RELATION='EMP')
The set operation MINUS is now done by the RDBI.
```

Example 14: A join operation may be required to return values selected from more than one relation. The names of all employees and the locations where they work may be listed by the query:

```
SEQUEL:
   SELECT EMP.NAME,DEPT.LOC
   FROM   EMP,DEPT
   WHERE  EMP.DNO=DEPT.DNO
```

```
DBC Command:
   RETRIEVE:(NAME,DNO)(RELATION='EMP')
            CONNECT ON (DNO,DNO)
            (LOC,DNO)(RELATION='DEPT')
```

Here, there are two field specification lists: (NAME,DNO) for the first query and (LOC,DNO) for the second query. The command is to connect (join)

on the two DNO attributes and return as response data triples of the form (NAME,DNO,LOC), where NAME is taken from the first field specifications list, LOC is taken from the second list, and DNO is common to both. The RDBI now returns to the user only the pairs (NAME,LOC) by deleting DNO from the triples returned by the DBC.

Example 15: In some circumstances, it is necessary to join a relation with itself according to some criterion. The relation name may then have to be listed more than once and labelled, e.g., X and Y may be two labels for a relation EMP. As an example, the following SEQUEL query will list the employee's name and his manager's name for each employee whose salary exceeds his manager's salary.

```
SEQUEL:
   SELECT X.NAME,Y.NAME
   FROM   EMP X, EMP Y
   WHERE  X.MGR=Y.EMPNO
   AND    X.SAL>Y.SAL

DBC Command:
   a.  RETRIEVE:(MGR)(RELATION='EMP')
                 CONNECT ON (MGR,EMPNO)
                 (EMPNO)(RELATION='EMP')
```

The only difference between this command and the command for Example 14 is that only one attribute is returned, instead of X.NAME and Y.NAME as well. This is because the AND clause has still got to be considered. Notice that since a manager has at least one employee (in general), a modified command (a') would also have the same effect as (a), yet taking less time to execute. However, (a') is not general enough for all situations.

```
   a'. RETRIEVE:(UNIQUE MGR)(RELATION='EMP')
```

For each manager number 'mi' returned by (a), do the following: Send a command

```
   b.  RETRIEVE:(NAME,SAL)((RELATION='EMP')&(EMPNO='mi'))
```

and for each (nj,sk) pair returned by (b), send a command

```
   c.  RETRIEVE:(NAME)((RELATION='EMP')&(MGR='mi')&(SAL>sk))
```

Notice that the name retrieved by (c) is an employee name, and that returned by (b) is the corresponding manager's name.

Steps (b) and (c) have been written in such a way that for every manager, the DBC accesses all his employees at the same time. These two steps could otherwise have been written such that for every employee, the DBC accesses all his managers at the same time. But, of course, every employee has a single manager. Therefore, the way we have written the commands is better than its alternative, since fewer number of accesses is required in the former case. The decision is made on the basis of

the fact that there are fewer unique values of MGR than there are of EMPNO.

Example 16: SEQUEL permits a label to be used to qualify attribute names outside the block in which the label is defined. The following query uses this feature in listing the suppliers who supply all the parts used by Dept. 50.

```
SEQUEL:
   SELECT SUPPLIER
   FROM   SUPPLY X
   WHERE
          (SELECT PART
          FROM    SUPPLY
          WHERE   SUPPLIER=X.SUPPLIER)
   CONTAINS
          (SELECT PART
          FROM    USAGE
          WHERE   DNO=50)
```

DBC Command:

a. RETRIEVE:(UNIQUE SUPPLIER)(RELATION='SUPPLY')
b. RETRIEVE:(PART)(RELATION='USAGE')&(DNO=50))

Since the block after CONTAINS has a comparison involving a constant, it needs to be executed only once. This is done by command (b) given above. For each supplier 'si' retrieved by (a), a DBC command

c. RETRIEVE:(PART)((RELATION='SUPPLY')&(SUPPLIER='SI'))

is sent, and the sets retrieved by (b) and (c) are compared by software.

The same query could have been made in SEQUEL by means of GROUP BY and the special function SET, as given below:

```
SELECT SUPPLIER
FROM   SUPPLY
GROUP  BY SUPPLIER
HAVING SET(PART) CONTAINS
       SELECT PART
       FROM   USAGE
       WHERE  DNO=50
```

The DBC commands would be the same as before.

Example 17: As a final example, consider listing the names of employees who have the same job and salary as those of Smith. This is done as follows:

```
SEQUEL:
   SELECT NAME
   FROM   EMP
   WHERE  <JOB,SAL> =
          SELECT JOB,SAL
          FROM   EMP
          WHERE  NAME='SMITH'
```

DBC Commands:

a. RETRIEVE:(JOB,SAL)((RELATION='EMP')&(NAME='SMITH'))

For the tuple (j,s) retrieved by (a), a second DBC command is sent:

b. RETRIEVE:(NAME)((RELATION='EMP')&(JOB='j')&(SAL='s'))

5.2 Use of the Clustering Information

Through an extensive list of examples, we have tried to indicate in the last section some of the principles involved in translating SEQUEL queries to DBC commands. The relation name associated with a query is seen to be a part of all DBC commands, thereby making it possible to limit the search to at most as many MAUs as are required to store the entire relation. This is because the DBC records are clustered primarily by the relation names. Not all record-contiguity information, however, has been made use of in formulating the DBC commands.

As we might recall from Section 4, the definition of a relational database consists of, besides other things, a specification of certain access aids such as the images and links. Furthermore, some of these access aids are declared to be applicable for clustering purposes. There can be no more than one clustering image or clustering link for any relation. We have shown in Section 4, how a special keyword with attribute CLUSTER is created for each DBC record that represents a tuple of a relation having a clustering image or clustering link. This keyword is created during the time of insertion of a new record into the database. Thereafter, whenever records are to be retrieved in response to a SEQUEL query, the special keyword may be regenerated and used as a predicate in the corresponding DBC query. This will be possible, however, only when the SEQUEL query consists of comparisons involving an attribute that is one of the attributes in a clustering image or clustering link.

Given a SEQUEL query, the DBC commands are created in two steps:

1. The DBC commands are initially created as shown in all our earlier examples.
2. For each DBC command thus generated, whenever possible, the special keyword <CLUSTER,cluster number> is computed and included as a predicate in the DBC command. The modified command is finally sent to the DBC for actual execution.

Computation of the special keyword is quite straightforward. If there is a clustering link on attributes A1,A2,...,An and if these same attributes occur in a DBC command (created in the first step given above) in the form of equality predicates, then the values (of the attributes) are hashed to generate a cluster number. The hashing algorithm is the same as the one used during the record-insertion process. As an example, consider that there is a clustering link as defined below:

```
CREATE CLUSTERING LINK L5
FROM   DEPT(DNO)
TO     EMP(DNO)
```

Then the EMP relation is clustered secondarily by DNO. Now assume that there is a SEQUEL query

```
SELECT EMPNO,NAME,JOB
FROM   EMP
WHERE  DNO=50
```

This query is translated to DBC commands in two steps:

Step 1. The initial DBC command generated is

```
RETRIEVE:(EMPNO,NAME,JOB)
         ((RELATION='EMP')&(DNO=50))
```

Step 2. The department number 50 is hashed to compute a cluster number i. The modified DBC command now generated is:

```
RETRIEVE:(EMPNO,NAME,JOB)
         ((RELATION='EMP')&(DNO=50)&(CLUSTER='i'))
```

In the case of a clustering image, the clustering table (discussed in Section 4) used during the process of record insertion is again used for retrieval purposes. For example, consider the clustering image as defined below:

```
CREATE CLUSTERING IMAGE I3
       ON EMP(DNO,SAL)
```

Let the possible department numbers be D100,D200,D300 and D400. Let us further assume that the SAL attribute (which has a very large range) is partitioned into five subranges. Then the clustering table may look like as shown in Figure 5.1. There are a total of 20 clusters for the EMP relation as shown in Figure 5.2. Now assume that there is a SEQUEL query:

```
SELECT EMPNO
FROM   EMP
WHERE  DNO='D200'
AND    COMM>5000
```

Then the DBC command is generated in the following steps:

Step 1. Generate initial command

```
RETRIEVE:(EMPNO)
   ((RELATION='EMP')&(DNO='D200')&(COMM>5000))
```

Step 2. Since DNO='D200' corresponds to the five cluster numbers 6,7,8,9 and 10, the modified DBC command is

```
RETRIEVE:(EMPNO)
   ((RELATION='EMP')&(DNO='D200')&(COMM>5000)
   &(CLUSTER≥6)&(CLUSTER≤10))
```

Attribute	DNO	SAL
Range	Small	Large
Number of values	4	5
Value partitions	D100	≤10,000
	D200	≤15,000
	D300	≤20,000
	D400	≤25,000
		>25,000

Figure 5.1. Table used for the clustering image on DNO and SAL attributes of the EMP relation

Cluster Number	DNO	SAL
1	D100	≤10,000
2	D100	≤15,000
3	D100	≤20,000
4	D100	≤25,000
5	D100	>25,000
6	D200	≤10,000
⋮	⋮	⋮
10	D200	>25,000
11	D300	≤10,000
⋮	⋮	⋮
15	D300	>25,000
16	D400	≤10,000
⋮	⋮	⋮
20	D400	>25,000

Figure 5.2. Cluster numbers corresponding to the various values of DNO and SAL attributes

5.3 Translating the Data Manipulation Statements

The data manipulation facilities of SEQUEL allow the user to directly change values in the database. Using these facilities, the user can insert, delete or update a tuple or a group of tuples in the database. He can also assign the result of a query to a newly-created relation.

A. Insertion

Any time after a relation has been defined, a new tuple may be inserted into the database by an INSERT statement. As an example, the following SEQUEL statement will insert a new employee named 'Jones' with employee number 535 in Dept. 51, having other attributes null:

```
INSERT INTO EMP(EMPNO,NAME,DNO):
            <535,'JONES',51>
```

To insert such a tuple, the RDBI first determines if the relation has a clustering link or a clustering image. Accordingly, it then creates a keyword <CLUSTER,cluster number> and then sends the DBC command

```
INSERT:(<RELATION='EMP'>,<EMPNO=535>,<NAME='JONES'>,
        <DNO=51>,<CLUSTER=cluster number>)
    with primary clustering condition (RELATION='EMP')
    and secondary clustering condition
              (CLUSTER=cluster number)
```

In case the secondary clustering condition cannot be determined either due to the absence of a clustering image or clustering link, or due to the absence of clustering attributes in the tuple to be inserted, then only the primary clustering condition is sent as part of the DBC command.

Set-oriented insertions in SEQUEL consist of the evaluation of a query and insertion of the resulting tuples into some existing relation. Assume, for example, that the database contains a relation called CANDIDATES which has columns for employee number, name, department number and salary. The following SEQUEL statement will then add to the CANDIDATES table all those employees whose commission is greater than half their salary:

```
INSERT INTO CANDIDATES:
  SELECT EMPNO,NAME,DNO,SAL
  FROM   EMP
  WHERE  COMM>0.5*SAL
```

The RDBI executes this statement in four steps:

(1) The query is evaluated by using the DBC RETRIEVE command.(Part

of the query is, of course, evaluated in software since the arithmetic operation 0.5*SAL cannot be done by the DBC hardware.)

(2) For each tuple retrieved in Step 1, the keyword <CLUSTER,cluster number> is generated, if the definition of the CANDIDATES relation allows it.

(3) For each tuple, a DBC record is formed as shown in Section 4. An INSERT command is now sent to the DBC for each DBC record created in Step 3. The primary and secondary clustering conditions are sent together with every record.

B. Deletion

Deletion, in SEQUEL, is done by means of a DELETE statement accompanied by a WHERE clause. The WHERE clause specifies the conditions that must be satisfied by the records to be deleted. A simple example of the DELETE operation is the deletion from the EMP relation the employee with employee number 561. The SEQUEL statement to achieve this is:

```
DELETE EMP
WHERE  EMPNO=561
```

The corresponding DBC command is:

```
DELETE:(RELATION='EMP')&(EMPNO=561)
```

Once again, whenever applicable, the cluster number to which the record(s) belongs may be specified by means of another predicate.

A more complex example that uses labels is the deletion from the DEPT relation all the departments having no employees. A SEQUEL statement to do the job is given below:

```
DELETE DEPT X
WHERE
       (SELECT COUNT(*)
       FROM    EMP
       WHERE   DNO=X.DNO)=0
```

The RDBI achieves the same effect with the following procedure:

(1) Send a DBC command

RETRIEVE:(DNO)(RELATION='DEPT')

For every department number 'di' retrieved by (1), do the following steps:

(2) Send a DBC command

```
RETRIEVE:[COUNT ONLY]
    ((RELATION='EMP')&(DNO='di'))
```

(3) If the result of (2) is zero, then send a command

DELETE:((RELATION='DEPT')&(DNO='di'))

C. Update

Updating a tuple in SEQUEL is done by an UPDATE statement with a SET clause specifying the updates to be made on the selected tuples. The RDBI can translate the UPDATE statements in one of two ways: 1) By using the RETRIEVE command to determine the MAU addresses of the selected records, and then using the REPLACE command to modify these records one at a time, or 2) By using the REPLACE command to modify all the selected records simultaneously. Which of these two methods is to be used depends on the actual SEQUEL statement. If the SET clause makes identical changes to all the selected tuples, only then can the second method be used. We illustrate the two cases with two examples.

Update Example 1: Update the EMP table by giving a 10% raise to all those employees who are in the CANDIDATES relation.

```
SEQUEL:
  UPDATE EMP
  SET    SAL=SAL*1.1
  WHERE  EMPNO  IN
         SELECT EMPNO
         FROM   CANDIDATES
```

DBC Commands:

a. RETRIEVE:(EMPNO)(RELATION='CANDIDATES')

For each employee number 'ei' retrieved by (a) send a command

b. RETRIEVE:[WITH POINTER] (SAL)
((RELATION='EMP')&(EMPNO='ei'))

Finally, for each pointer 'pi' and salary 'sj' retrieved by (b), compute a new salary sk=(1.1*sj) and send a DBC command

c. REPLACE:(pi)(<SAL = sk>)

This command specifies that the record pointed to by 'pi' is to get the modified value 'sk' for the SAL attribute.

Update Example 2: Update the EMP relation by giving a commission of $5000 to every clerk.

```
SEQUEL:
  UPDATE EMP
  SET    COMM=5000
  WHERE  JOB='CLERK"
```

DBC Command:

```
REPLACE:((RELATION='EMP')&(JOB='CLERK'))
          (<COMM,5000>)
```

This command specifies that all the records satisfying the query conjunct

must have the value of their COMM attribute modified to 5000.

D. Assignment

An assignment statement in SEQUEL allows the result of a query to be copied into a newly-created relation in the database. Thus, the execution of an assignment statement by the RDBI is done in two parts:

(1) The records satisfying the query are retrieved as shown in Section 5.1, and

(2) A new relation is created with the records retrieved in (1). These records are then stored in the database.

6. RELATIONAL DATA CONTROL FACILITIES

System R, our model for a relational database management system, has extensive data control facilities that enable users to control access to their data by other users, and to exercise control over the integrity of data values. The data control facilities have four aspects: transactions, authorization, integrity assertions and triggers.

A transaction is a series of statements which the user wishes to be processed as an atomic act. The user controls transactions by the operators BEGIN-TRANS and END-TRANS. The user may specify save points within a transaction by the operator SAVE. As long as a transaction is active, the user may back up to the beginning of the transaction or to any internal save point by the operator RESTORE. Implementation of transactions on the DBC involves no new concepts. From the save point onwards, any update made by the transaction will cause the old version of the updated record to be stored in the DBC mass memory as part of a temporary database. A subsequent RESTORE command may be executed by simply deleting from the database all updated records and replacing them by their old copies stored away in the temporary database.

6.1 Views

System R relies on its view mechanism for authorization. As opposed to the base relations which are physically stored in the database, a view is a virtual relation which is a dynamic'window' on the database. In response to a query, the tuples of a view are dynamically computed from the base relation(s). An update to a view is not allowed if it is defined on more than one base relation. We shall briefly describe below how a view is defined in SEQUEL and how it is implemented in the DBC.

Any SEQUEL query which results in a relation may be used to define a view. The RDBI translates any SEQUEL query by first translating it into DBC commands in the normal way (as discussed in Section 5). Any reference to a view, in the first step, is treated as a base relation, but the DBC commands thus formed are only intermediate commands not to be immediately transmitted to the DBC. These commands are now qualified with predicates determined from the view definition. The modified commands, then, are the ones that are transferred to the DBC for execution. We shall illustrate this process by means of the three most important cases of view definition.

A view may be a row and column subset of a base relation. For example, the following SEQUEL statement may be used to define a view called D50 containing the employee number, name and job of the employees who work in Dept. 50.

```
DEFINE VIEW D50 AS:
  SELECT EMPNO,NAME,JOB
  FROM   EMP
  WHERE  DNO=50
```

A user may now wish to find the names of all clerks in D50 by issuing the SEQUEL statement

```
SELECT NAME
FROM   D50
WHERE  JOB='CLERK'
```

This statement is first translated into an intermediate DBC command

```
RETRIEVE:(NAME)((RELATION='D50')&(JOB='CLERK'))
```

However, there is no stored relation called D50. Hence, the view definition for D50 is used to replace the predicate (RELATION='D50') by ((RELATION='EMP')& (DNO='D50')). We thereby come up with the modified DBC command

```
RETRIEVE:(NAME)
       ((RELATION='EMP')&(DNO=50)&(JOB='CLERK'))
```

This is the command that is finally transmitted to the DBC, after making sure that no reference is made to any field (in this case, NAME) that is not included in the view definition (in this case, NAME does occur in the definition of D50).

A view may also be the join of the information in two or more relations. A view called DS, for example, may be defined as follows as a join of the PART attribute of the relations USAGE and SUPPLY:

```
DEFINE VIEW DS AS:
   SELECT DNO,SUPPLIER
   FROM   USAGE,SUPPLY
   WHERE  USAGE.PART=SUPPLY.PART
```

To find the department numbers of the departments to which the supplier 'Jones' supplies any part, one may issue the following SEQUEL statement:

```
SELECT DNO
FROM   DS
WHERE  SUPPLIER='JONES'
```

The first step in the translation of this statement is the creation of the DBC command

```
RETRIEVE:(DNO)((RELATION='DS')&(SUPPLIER='JONES'))
```

Since DS is only a view, the view definition is now used to come up with the DBC commands:

a. RETRIEVE:(PART)((RELATION='SUPPLY')&(SUPPLIER='JONES'))

and for each part 'pi' retrieved by (a), another command

b. RETRIEVE:(DNO)((RELATION='USAGE')&(PART='pi'))

A view may be a summary of the information in a base relation. For example, to define a view consisting of the average salary of each department, one can issue the SEQUEL statement:

```
DEFINE VIEW AVGSAL AS:
   SELECT AVG(SAL)
   FROM   EMP
   GROUP  BY DNO
```

Any reference to the average salary of some department 'di' in AVGSAL may then be translated to the DBC command

RETRIEVE:(AVG(SAL))((RELATION='EMP')&(DNO='di'))

Although views are used for facilitating the description of a relational query, a more important use of the views is in the process of authorization. The creator of a base relation may grant (and may later revoke) any privilege (such as READ, INSERT, DELETE, UPDATE) to (from) other users. He may further provide any of these users with the GRANT option. In that case, the latter user has the privilege to grant (revoke) all his privileges on the given relation to (from) yet another user. If a user is authorized to create a view on a base relation then he has the sole authority to perform any action on it consistent with his privileges on the base relation. He may also grant those privileges to another user.

6.2 Authorization and Security

System R allows for an extremely simple method of authorization checking [15]. System R maintains two tables for the use of the authorization subsystem, namely, SYSAUTH and SYSCOLAUTH. The SYSAUTH table has upto two rows for each combination of relation (base relation or view) and user. The columns in the SYSAUTH table correspond to user id, base relation or view name, type (whether base relation or view), a column for each of the privileges on the relation (the entry being 'Y' or 'N' in every such column) and a column for grant option ('Y' or 'N'). For each relation on which a user is authorized to perform some action, there are upto two tuples in SYSAUTH: one for grantable and the other for non-grantable privileges.

In case the user has update rights on a relation, the table SYSCOLAUTH indicates precisely those columns of the relation on which the user has the update privilege. These two tables SYSAUTH and SYSCOLAUTH are updated whenever a new base relation or view is created or whenever an authorized user executes a GRANT statement thereby granting a set of privileges to one or more other users. The two tables are referenced immediately before the execution of any SEQUEL statement. Since all SEQUEL statements refer to an action and to one or more relations, authorization checking can be accomplished by the RDBI even before the statements are translated to DBC commands. The checking process is exactly identical to that in System R.

As a possibility, however, an extra dimension can be added to the authorization mechanism of System R by using the hardware security enforcement feature of the DBC. Although the view mechanism of System R is important for its use in specifying authorization, a view need not necessarily be used for that purpose alone. A view may be defined, for example, for the sole purpose of simplifying SEQUEL queries. Therefore, it may not be practical to represent every view of a base relation in the security module of the DBC. Furthermore, this is not necessary since security enforcement by views and relations can be easily done even before the SEQUEL statements are translated to DBC commands.

Suppose, now, that many different row-subsets of a relation are to be authorized for access by various users. Then a view must be defined for every such subset, and each user must be aware of the names of the views corresponding to the row-subsets he is allowed to access. A better scheme would be to somehow allow each user access to appropriate logical subsets of the base relation without requiring the user to remember the names of these subsets. In any SEQUEL statement, the user can then make a reference directly to the base relation. If the statement requires the retrieval (or update) of any part of the relation on which he has no authorization for the corresponding action, then the statement will not be executed.

Such a scheme is ideally implemented on the DBC. The values of any attribute to be used for authorization may be partitioned. Access privileges of any user may be specified in terms of predicates using the above attributes. The specifications of the access privileges are stored in the DBC structure memory. Users may then write SEQUEL statements that refer to an entire base relation. These statements will be converted to DBC commands, and the commands will be executed by the DBC only if the

authorization specifications of the user determined automatically by the DBC allow such operations.

Here is a simplified exposition of how the DBC security mechanism works. The creator of a file may specify certain attributes as security attributes. The file creator also specifies security descriptors that partition the values of the security attributes.

The capabilities of a user are specified in the form of file sanctions, a file sanction being a pair <predicate conjunct, access privilege>. Each predicate in a file conjunct is restricted such that it must refer to complete blocks of the partition made by the corresponding security descriptors (i.e., the descriptors whose attribute parts, agree with the attribute part of the predicate). In addition to the file sanctions, the user capability also consists of certain default access privileges applicable to any record that does not satisfy the predicate conjunct in any file sanction.

The DBC will then create an atomic access privilege list (AAPL) for the given user. An atom is a set of records consisting of keywords that satisfy the exactly same security descriptors. Using the user capabilities and the security descriptors of the file, the DBC creates an AAPL for each user. An AAPL consists of entries of the form <atom number, access privilege>. The construction of an AAPL is done only once, i.e., one AAPL is constructed for each <file,user> pair.

During the actual command execution time, the DBC refers to the AAPL corresponding to the user who makes the command and the file to which the command refers. For every predicate conjunct within the command, the DBC determines the atoms to which it will refer. Using the AAPL, the DBC now decides whether the requested access is grantable or not. Only if the access is grantable for every referred atom is the command finally carried out.

As an example, consider that there is a file with two security attributes, JOB and SALARY. The security descriptors are as given below:

SD1. 0<SALARY≤5000

SD2. 5000<SALARY≤10000

SD3. 10000<SALARY≤30000

SD4. 30000<SALARY≤10000000

SD5. JOB='CLERK'

SD6. JOB='ANALYST'

SD7. JOB='MANAGER'

Further assume that the capability of a particular user is given as:

	File Sanction	Access Privilege
(1)	(SALARY≤10000)	READ,UPDATE,DELETE
(2)	(SALARY>10000)&(JOB='ANALYST')	READ,UPDATE
(3)	Default	READ

The DBC now creates an AAPL for the given user. There are 12 atoms and they have the following access privileges:

(1)	SD1,SD5	READ,UPDATE,DELETE
(2)	SD1,SD6	READ,UPDATE,DELETE
(3)	SD1,SD7	READ,UPDATE,DELETE
(4)	SD2,SD5	READ,UPDATE,DELETE
(5)	SD2,SD6	READ,UPDATE,DELETE
(6)	SD2,SD7	READ,UPDATE,DELETE
(7)	SD3,SD5	READ
(8)	SD3,SD6	READ,UPDATE
(9)	SD3,SD7	READ
(10)	SD4,SD5	READ
(11)	SD4,SD6	READ,UPDATE
(12)	SD4,SD7	READ

A DBC command made by this user can now be compared against his AAPL to determine whether the required access is to be granted. For example, if the command is to update some field(s) of any record belonging to an analyst earning more than 12000, then the atoms referred to are atom 8 (which satisfies SD3 and SD6) and atom 11 (which satisfies SD4 and SD6). The update privileges are grantable for both these atoms. Therefore, the command is carried out. On the other hand, if the command is to update some field(s) of any clerk earning between 6000 and 15000, then the atoms referred to are atom 4 (which satisfies SD2 and SD5) and atom 7 (which satisfies SD3 and SD5). While the update privilege is grantable for atom 4, it is not grantable for atom 7. Therefore, the given command is not carried out, i.e., the requested access is denied.

6.3 Assertions and Triggers

Another important aspect of data control as provided in System R is that of assertions about data integrity. Any SEQUEL logical expression (e.g., AVG(SAL)>20000) associated with a base relation or view may be stated as an integrity assertion. At the time an assertion is made (by an ASSERT

statement), its truth is checked; if true, the assertion is enforced until it is explicitly dropped (by a DROP ASSERTION statement). As examples of the implementation of assertions on the DBC, consider the following:

Example 1: No employee should have a salary greater than 50000. In SEQUEL, this assertion is written as:

ASSERT ON EMP: SAL ≤ 50000

In the DBC, this assertion is first checked to be true by means of the command

RETRIEVE:[COUNT ONLY]((RELATION='EMP')&(SAL>50000))

If the count is zero, then the assertion is true to start with. Thereafter, after any update to employee records, the SAL field is checked to verify that the truth of the assertion still holds.

Example 2: The salary of an employee should never decrease. This is written in SEQUEL as:

ASSERT ON UPDATE TO EMP: NEW SAL ≥ OLD SAL

This assertion requires no initial chekcing. Only during an update on the EMP relation, must it be checked if the SAL field is going to be modified. In such a case, an extra retrieval query may be required by the DBC in order to retrieve the records that are going to be affected by the update; the assertion is checked via software, and then the update is made if the assertion holds.

A final data control aspect of System R is the concept of triggers, which is a generalization of the concept of assertions. A trigger causes a prespecified sequence of SEQUEL statements to be executed whenever some triggering event occurs, such as retrieval, deletion, insertion or update of a base relation or view. The RDBI can monitor such events by simply scanning a transaction for SEQUEL statements that correspond to a particular triggering event. Immediately after each of these statements, a call statement is included to invoke the appropriate trigger routine. The modified transactions and the trigger routines are now translated, as usual, into DBC commands.

7. PERFORMANCE ANALYSIS

Because of the parallelism involved in the operations performed by the DBC, it should be intuitively clear that user transactions will run faster on the DBC than on a conventional computer. The speed is further enhanced by the fact that a sequence of software operations can be replaced completely by a single DBC command. For example, in order to find all the records satisfying a conjunct of predicates, a conventional system will first determine (in some manner,e.g., via an index) the eligible records. It will then retrieve these records and compare each of them against the given predicates. In the DBC, on the other hand, not only are all the eligible records retrieved in parallel, but it is also true that this set of retrieved records is exactly the required response set. The reason is simply that records are compared against the given predicates simultaneously with their retrieval, thereby rendering unnecessary any subsequent software refinement of the retrieved set.

In this section, we shall make an analytical study of the DBC performance and compare it against that of a conventional computer where a relational database management system (in particular, System R) is being supported. The DBC will function in conjunction with a front-end computer. The support software, which we have been calling the Relational Data Base Interface(RDBI) is housed in the front-end computer. The RDBI interprets the data management calls of user programs and executes them with the aid of the DBC. We shall call the environment consisting of the DBC and front-end computer as the DBC environment. A conventional system, on the other hand, consists only of a general-purpose computer (GPC) which houses the database management system software and executes user transactions by reading (writing) record from (to) conventional secondary storage devices. We shall call such an environment a GPC environment.

Although the response time and throughput of a system is the most widely used measure of performance of the system, the cost of the system is surely a complementary measure since a tradeoff is involved between the cost and speed. We, therefore, will start with estimating the cost of a system by determining the storage requirement. The cost of the other components of a system will not be considered since they are fixed (e.g., the DBC is itself a fixed-cost component in a DBC environment). We will then go on to measure response time in terms of the number of secondary storage accesses and processor time required to complete the execution of user transactions. Numerical results are also computed for typical database parameters.

7.1 Mass Storage Requirement

The mass memory of the DBC stores the database records. Correspondingly, the secondary storage of a conventional relational system stores the tuples. Here, in this section, we shall try to estimate this storage requirement. The following definitions will be used:

- n = relation cardinality (# of tuples or records in the relation);
- d = degree of a relation (# of fields);
- p = length of a pointer field (or Tuple Identifier, TID), in number of bytes;
- ℓ = # of links defined on a relation;
- v_i = average length, in bytes, of the value of the i-th attribute of a relation; and
- a_i = average length, in bytes, of the i-th attribute name of a relation.

We will ignore in our analysis the details of certain implementation features since they are non-standard and have only marginal effect on the storage requirement. For example, the loading factor of the physical blocks may be maintained at a level slightly below unity in order to allow for database growth. Even if the loading factor were to have any appreciable effect on the storage requirement of a given system, this effect will be nullified when we compare two different systems against one another.

GPC Environment

In a conventional implementation of System R, every physical tuple consists of an ordered list of values and a pointer (or TID field) for every link defined on the relation to which the tuple belongs. Thus, the mass storage requirement, M_g, for a given relation is

$$M_g = n(\sum_d v_i + p\ell).$$

In case, $v_i = v$, for every i, we have

$$M_g = n(vd + p\ell).$$

DBC Environment

For each relation of cardinality n, the DBC stores n records. A record is composed of d attribute-value pairs if the degree of the relation is d. A record also contains a special keyword with attribute RELATION to identify the relation to which it belongs. In addition, assuming that a clustering link or a clustering image has been defined on the relation, another special keyword

with attribute CLUSTER is also included in the record. Thus, the mass storage requirement, M_d, for any given relation is

$$M_d = n \sum_{d+2} (v_i + a_i).$$

where the two special keywords are numbered (d+1)-th and (d+2)-th, respectively.

Since the DBC assigns a unique fixed-length code to each attribute, it follows that $a_i = a$, for every i. We, therefore, have

$$M_d = na(d+2) + n \sum_{d+2} v_i.$$

Further, if for every i, $v_i = v$, we have

$$M_d = n(d+2)(v+a).$$

We now define the mass storage ratio R_m as the ratio of mass storage requirements in the two different environments, namely, the GPC environment and the DBC environment. Therefore,

$$R_m = M_g/M_d = (\sum_{d} v_i + p\ell)/(\sum_{d+2} v_i + (d+2)a)$$

If $v_i = v$ for every i, we have

$$R_m = (vd + p\ell)/((d+2)(v+a))$$

In Figure 7.1, we have tabulated the mass storage ratio R_m for p = 4 bytes, a = 2 bytes and various values of v, ℓ and d. Since the number of attributes in a file is small, a length of 2 bytes for attributes name should be sufficient. The average length of the value part of an attribute-value pair is varied in steps of 2, from 2 to 8. Since the number of links defined on a relation is not likely to exceed the number of attributes (unless an attribute appears in a number of links, each connecting two relations), we may assume that in a practical database, $\ell \leq d$. Thus, we notice from Figure 7.1 that R_m is usually less than unity. Furthermore, since the number of links defined on a relation is usually one or more, the value of R_m is likely to be greater than 0.5. That is, in a practical database,

$$0.5 \leq R_m \leq 1.0$$

We, therefore, conclude that the mass memory requirement in a DBC environment is somewhat more than and up to double the requirement in a GPC environment.

7.2 Directory Storage Requirement

While the mass memory stores the database files containing the tuples or records, storage is also required for keeping the indexes (directories)

ℓ \ d	2	3	4	5	10
0	0.25	0.30	0.33	0.36	0.42
1	0.50	0.50	0.50	0.50	0.50
2	0.75	0.70	0.67	0.64	0.58
3	-	0.90	0.83	0.79	0.67
4	-	-	1.00	0.93	0.75
5	-	-	-	1.07	0.83

(i) v=2,p=4,a=2

ℓ \ d	2	3	4	5	10
0	0.33	0.40	0.44	0.48	0.56
1	0.50	0.53	0.56	0.57	0.61
2	0.67	0.67	0.67	0.67	0.67
3	-	0.80	0.78	0.76	0.72
4	-	-	0.89	0.86	0.78
5	-	-	-	0.95	0.83

(ii) v=4,p=4,a=2

ℓ \ d	2	3	4	5	10
0	0.38	0.45	0.50	0.54	0.63
1	0.50	0.55	0.58	0.61	0.67
2	0.63	0.65	0.67	0.68	0.71
3	-	0.75	0.75	0.75	0.75
4	-	-	0.83	0.82	0.79
5	-	-	-	0.89	0.83

(iii) v=6,p=4,a=2

ℓ \ d	2	3	4	5	10
0	0.40	0.48	0.53	0.57	0.67
1	0.50	0.56	0.60	0.63	0.70
2	0.60	0.64	0.67	0.69	0.73
3	-	0.72	0.73	0.74	0.77
4	-	-	0.80	0.80	0.80
5	-	-	-	0.86	0.83

(iv) v=8,p=4,a=2

Figure 7.1 Mass storage ratio R_m for various values of ℓ, d, v, p, and a

and the database definition. The database definition consists of the characteristics of every relation such as relation name, degree, attribute names and types, names and definition of links and images, and definition of triggers and assertions. The database definition constitutes the conceptual view or schema of the database. It must be stored by any system that implements the given database. Thus, the storage requirement for the schema is independent of the machine on which the database management system is being implemented. We shall,therefore, make no further attempt to estimate the memory requirement for the schema.

More important is the amount of memory occupied by the indexes. The size and structure of the indexes varies from one realization of the database to another depending on the machine which supports the database. This is particularly true when one machine uses conventional location-addressed secondary storage and the other is a database machine using (partitioned) content-addressable memory and having the capabilities for hardware maintenance of directories. We shall now analyze the directory storage requirement in the two different cases.

GPC Environment

In System R, each image is an index to a relation. System R's Relational *Storage System (RSS) maintains* each image through the use of a multi-page index structure. Each index is a dense index in the sense that every value of the underlying attribute or attribute combination is represented in the index, thereby making it possible to determine the address of every record satisfying an equality predicate based on the above attribute or attribute combination. The pages for a given index are organized into a balanced hierarchy structure, called B-trees [16].

A B-tree of order s is a tree which satisfies the following properties:

(1) Every node has $\leq$ s sons.

(2) Every node, except for the root and the leaves has $\geq$ m/2 sons.

(3) The root has at least 2 sons.

(4) All leaves appear on the same level, and carry no information.

(5) All non-leaf nodes with k sons contain (k-1) keys.

In Figure 7.2, we have extracted from [16] an example B-tree of order 7, with the root at level 0 and with all leaves at level 3.

In System R, the B-trees for the images slightly differ from the above definition. Since the records are not actually stored in the B-trees, their addresses (or TIDs) must be stored. Thus the leaves are not empty; they carry

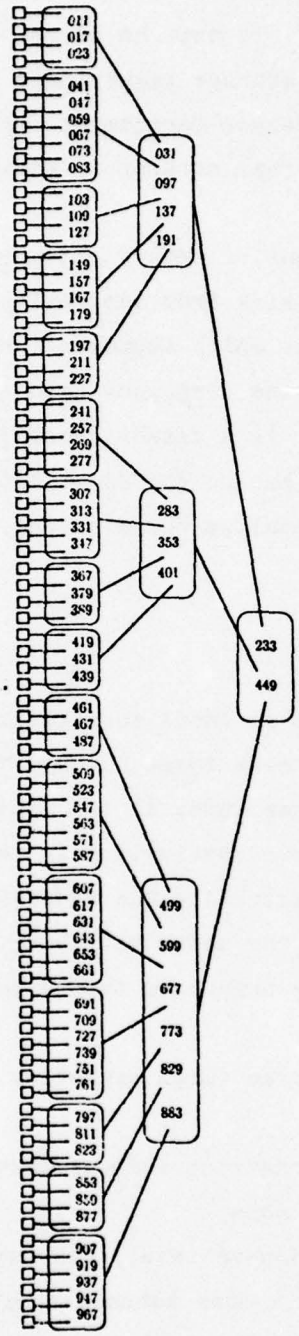


Figure 7.2 A B-tree of order 7

a list of TIDs. The B-trees in System R are defined as follows. Each page is a node within the tree and contains an ordered sequence of index entries. For each non-leaf node , an entry consists of a <sort value,pointer> pair. The pointer addresses another page in the same structure which may be either a leaf page or another non-leaf page. In either case, the target page contains entries for sort values less than or equal to the given one. For the leaf nodes, an entry is a combination of sort values along with an ascending list of TIDs for tuples having exactly those sort values. The leaf pages are chained in a doubly-linked list, so that sequential access can be supported from leaf to leaf. The structure of the nodes in such a B-tree is depicted in Figure 7.3.

To compute the storage requirement per image we use the following nomenclature:

- n = relation cardinality (# of tuples in the relation);
- p = length of an internal pointer, in bytes;
- t = length of a TID, in bytes;
- v = average length (in bytes) of a value of the attribute on which the image is defined. We assume that the image is a single-attribute image, since this is the most common case;
- s = order of the B-tree. The order, which determines the range of the number of pointer (and key) fields in each non-leaf page (internal node), depends on page size, on average key length v, and on the length p of an internal pointer;
- i = image cardinality, which is the number of distinct sort field values in the image; and
- b_g = page size, in bytes.

For given parameters n,p,t,v,i and b_g we shall try to compute the minimum amount of memory required to store an image. We begin by computing the expected minimum number of external nodes in the B-tree. We then compute the order s of the B-tree. Next we compute the minimum number of internal nodes, thereby completing the analysis.

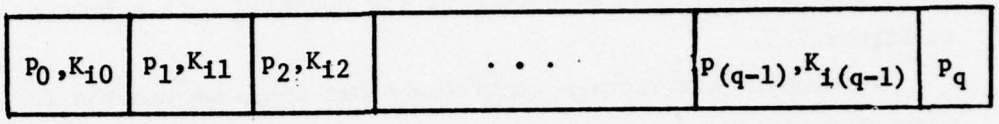
Since the average number of TIDs per key is n/i, we may expect

$$(b_g - 2p)/(v + (n/i)t)$$

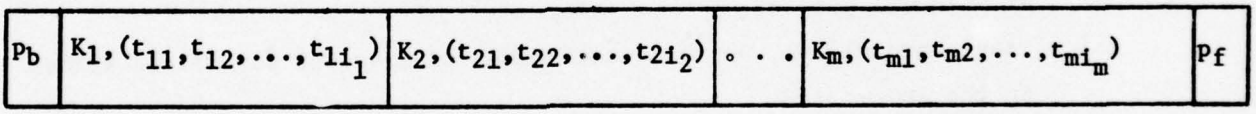
keys per external node. Hence, the minimum number of external nodes E is given by

$$E = \lceil i(v + (n/i)t)/(b_g - 2p) \rceil$$

$$= \lceil (iv + nt)/(b_g - 2p) \rceil$$



(i) Structure of an internal node in the B-tree



(ii) Structure of an external (leaf) node in the B-tree (p_b is a backward pointer to the preceding leaf page and p_f is a forward pointer to the next leaf page)

Figure 7.3 Nodes in a B-tree
(K stands for a key, t for
a TID and p for an internal pointer)

The order s of the B-tree, which is the maximum number of pointer fields in each internal node, is given by

$$s = \lfloor (b_g - p)/(v + p) \rfloor + 1.$$

In order to compute the minimum number of internal nodes, I, notice that there are E nodes in level u, where u is the maximum level of the tree; there are at least $\lceil E/s \rceil$ nodes in level (u-1), at least $\lceil E/s^2 \rceil$ nodes in level (u-2),..., at least $\lceil E/s^u \rceil$ nodes in level 0. Since there is only one node in level 0, it follows that

$$E/s^u \leq 1 < E/s^{u-1}.$$

Thus, $u = \lceil \log_s E \rceil$.

The minimum number I of internal nodes is now given by

$$\begin{aligned} I &= \lceil E/s \rceil + \lceil E/s^2 \rceil + \ldots + \lceil E/s^u \rceil \\ &\geq E(s^{u-1} - 1)/(s^{u+1} - s^u) \end{aligned}$$

In most practical situations s is large and, therefore, even if u is small (say, 2 or 3), $s^{u-1} \gg 1$. Hence,

$$I \simeq E/(s^2 - s).$$

Finally, the minimum directory storage requirement per image, D_g, is given by

$$\begin{aligned} D_g &= (E + I) \text{ pages} \\ &= (E + I)b_g \text{ bytes} \\ &= E(1 + (1/(s^2 - s)))b_g \text{ bytes} \end{aligned}$$

In the above calculations we have assumed the fact that every internal node has s pointers. Due to updates on the database, it is more likely that there will be approximately 0.75s pointers per internal node, since the number, in steady state, is likely to be uniformly distributed between 0.5s and s. The directory storage requirement will therefore be somewhat greater than what the above calculations indicate.

DBC Environment

Even though the RDBI maintains no directories corresponding to the images and links defined on relations, some minimal directories are, in fact, maintained in the structure memory of the DBC. We will now try to estimate the size of such directories.

To begin with, we may recall that there are directory entries for only two classes of keywords: those with attribute RELATION and those with attribute CLUSTER. Since these keywords are also defined to be clustering keywords, the DBC assigns a unique cluster number to all records having the same two keywords

<RELATION, r-name> and <CLUSTER, c-num>. Thus, a cluster in the DBC consists of the set of records S such that two records R1 and R2 are in S if and only if

<RELATION, r-name1>,<CLUSTER, c-num1> $\in$ R1,

<RELATION, r-name2>, <CLUSTER, c-num2> $\in$ R2,

r-name1 = r-name2, and c-num1 = c-num2 .

A directory entry in the DBC is of the following form (where we have ignored security atom numbers) :

<keyword, (index1, index2,..., index **h**) >

where each index is of the form :

(MAU#, cluster#)

We use the following nomenclature:

a = length of a (coded) attribute name, in bytes;

v = average length in bytes of the (coded) value part of the keywords with attributes RELATION and CLUSTER;

c = numbers of clusters of a relation (usually of the order of the number of MAUs required to store the relation);

m = length of an MAU#, in bytes;

k = length of a cluster#, in bytes; and

j = average number of MAUs spanned by a cluster (the number of MAUs spanned by a cluster is defined to be the number of MAUs in which there is at least one record belonging to the cluster).

The number of different index terms (i.e.,(MAU#,cluster#)pairs) for a relation is simply equal to cj. Since, for any given relation, there is only one directory keyword with attribute RELATION, the corresponding directory must have all the index terms for the relation. On the other hand, there are up to c directory keywords with attribute CLUSTER, and each of the corresponding entries has an average of j index terms. Thus, the directory memory requirements for a relation is given by

D_d = storage for the entry with keyword <RELATION, - >

+ storage for all entries with keywords of the form <CLUSTER, - >

$= ((a + v) + cj(m + k))$

$+ c((a + v) + j(m + k))$

$= (c + 1)(a + v) + 2cj(m + k)$.

We observe that the directory memory requirement per relation, D_d, of the DBC is independent of the total number of images defined on a relation. This contrasts with the fact that in a GPC environment the directory memory

requirement per relation is the sum total of the storage requirements for all images on a relation. If there are L images on a relation and each image requires the same space D_g, then the directory memory requirement per relation, in the GPC environment, is LD_g. We define the directory storage ratio R_d as the ratio of the directory memory requirement in the GPC environment to that in the DBC environment. If there are L images per relation and every image is of equal size, we then have

$$R_d = LD_g/D_d$$

$$\text{where } D_d = (c+1)(a+v) + 2cj(m+k),$$

$$D_g = E(1+(1/(s^2 - s)))b_g,$$

$$E = \lceil (iv + nt)/(b_g - 2p) \rceil,$$

$$\text{and } s = \lfloor (b_g - p)/(v+p) \rfloor + 1.$$

In the computation of D_d, the value of j, which is the number of MAUs spanned by a cluster, is a dependent parameter. It depends on the cluster size, MAU size, loading factor of the database and the storage pattern. We conducted a number of simulation experiments to estimate the value of j. For a given number of clusters c, it was assumed that there is an equal probability of any record belonging to any given cluster. The essential structure of each experiment is summarized below in the form of an algorithm:

Step 0. To start with, all MAUs and all clusters are empty.
Step 1. If enough records have been generated so that the loading factor of the mass memory is 1, then compute statistics and stop. Else go to 2.
Step 2. Generate a random record and determine its cluster number, CN. (It is actually enough to generate only a random cluster number CN, since all records are assumed to be of equal size).
Step 3. If the cluster CN was previously empty, then select an MAU which is currently being occupied by the least number of records. Assign that MAU to the cluster CN, store the record in that MAU and go back to 1.
Step 4. If the cluster CN is not empty, then it has already been assigned one or more MAUs, all of which, except one, is known to be full. Select from these MAUs, the only one MAU#, M, which possibly is still not full.
Step 5. If M is not full then store the record in that MAU and go back to 1. Else go to 6.
Step 6. Select an MAU which is currently being occupied by the least number of records. Add that MAU to the list of MAUs assigned to the cluster CN, store the record in that MAU and go back to 1.

A total of 40 simulation runs were made in all, for 100,000 fixed-length records in each case. An experiment was conducted for every combination of

(1) total number of MAUs, taken from the set {50,100,200,400,800},

(2) ratio of total number of clusters to the total number of MAUs, taken from the set {1,2,4,8} and

(3) loading factor, taken from the set {0.9,0.95}.

Notice that, since the number of MAUs, the number of records and the loading factor are constant for any given experiment, the length of the fixed-size records for that experiment is automatically determined (in terms of the size of an MAU). For example, if there are 100,000 records, 100 MAUs and the loading factor is 0.9, then the ratio

Record Size/MAU Size = 100 * 0.9/100000 = 0.0009

We observe from the tables in Figure 7.4 that even at a very high loading factor, 0.95, the value of j does not exceed 2. In practice, even when database updates are taken into account, as long as the loading factor does not exceed 0.95, say, we do not expect a cluster to span more than two MAUs, on the average. Therefore, we anticipate the following bounds on j

$$1 \leq j \leq 2.$$

Coming back to the computation of the directory storage ratio R_d, we assume that there is only one image per relation, i.e., L=1. Consider the following parameters:

a = length of a coded attribute name = 2 bytes;

j = # of MAUs spanned by a cluster = 2;

v = length of the value of an attribute = 4 bytes;

m = length of an MAU # = 4 bytes;

k = length of a cluster # = 4 bytes;

t = length of a TID = 4 bytes;

p = length of an internal pointer in a B-tree = 4 bytes;

b_g = page size (size of a node in the B-tree) = 4000 bytes;
(This is, perhaps, larger than usual; but, then, smaller b_g will only reduce D_g, thereby reducing the storage ratio);

r = ratio of the number of clusters of a relation to the number of MAUs occupied by the relation = 5;

b_d = MAU size = 500,000 bytes;

n = number of records in the relation, taken from the set {1000,2000, 5000,10000,20000,50000,100000};

n/i = ratio of relation cardinality to image cardinality, taken from the set {1,2,5,10,20,50,100}; and

q = length of a DBC record, in bytes, taken from the set {50,100,200,500, 1000,2000}.

M \ r	1	2	4	8
50	1.00	1.44	1.51	1.31
100	1.00	1.32	1.56	1.28
200	1.01	1.34	1.39	1.27
400	1.05	1.39	1.38	1.26
800	1.13	1.36	1.35	1.21

(i) $\ell = 0.9$

M \ r	1	2	4	8
50	1.00	1.33	1.53	1.56
100	1.06	1.43	1.70	1.44
200	1.14	1.48	1.49	1.42
400	1.21	1.53	1.48	1.37
800	1.33	1.52	1.46	1.30

(ii) $\ell = 0.95$

Figure 7.4 The number, j, of MAUs spanned by a cluster for various combinations of loading factor ℓ, number of MAUs M and ratio, r, of number of clusters to number of MAUs

Using the fact that the number of MAUs required for a relation is $\lceil nq/d \rceil$ and the fact that c, the number of clusters of the relation, is r times the above-mentioned number, we can now compute the directory storage ratio R_d. These calculations are tabulated in Figure 7.5. Observe that, other parameters remaining unchanged, after the number of records, n, has reached a high enough value, further increase in n does not have much effect, since both D_g and D_d tend to increase proportionately with n, for large n. Further observe that as DBC record length increases, fewer and fewer records are accommodated in an MAU, thereby increasing the number of index terms and hence the storage ratio R_d.

We notice that for a reasonable record length between 100 and 1000 bytes, the DBC directory memory requirement lies between 0.05% and 10% of that of a conventional system. Furthermore, if there are more than one image per relation (which is often the case), then the directory memory requirement in a GPC environment increases proportionately with the number of images. The DBC directory memory requirement, in contrast, remains steady.

7.3 Query Execution Time

Query execution time is perhaps the single most important measure of performance of a database management system. Given a SEQUEL query in a conventional GPC environment, the system first uses an optimizer to determine a good access strategy from among a large number of possible access strategies. Ignoring the parsing and optimization time, the execution time of a query consists mainly of

(1) the time to access a number of index pages and search their contents in order to determine a list of eligible TIDs,

(2) the time to access a number of data pages in order to fetch the eligible tuples, and

(3) the CPU time to determine the final response set from the list of eligible tuples.

For a given query, a single predicate of a predicate conjunct in the query may be used for determining the eligible TIDs. After the corresponding tuples are retrieved, they are placed in the final response set only if they satisfy all the other predicates in the predicate conjunct.

In a DBC environment, the execution time of a query consists mainly of

(1) hardware search time of the structure memory to determine the eligible MAUs, and

n \ n/i	1	2	5	10	20	50	100
1000	61.2	40.8	40.8	40.8	40.8	40.8	40.8
2000	102.0	81.6	61.2	61.2	61.2	61.2	61.2
5000	224.5	163.3	142.9	122.4	122.4	122.4	122.4
10000	428.6	326.5	265.3	244.9	224.5	224.5	224.5
20000	424.9	321.2	259.1	238.3	228.0	217.6	217.6
50000	422.6	318.0	255.2	234.3	221.8	217.6	213.4
100000	421.8	316.9	253.9	232.9	222.5	216.2	214.1

q = 50 bytes

n \ n/i	1	2	5	10	20	50	100
1000	61.2	40.8	40.8	40.8	40.8	40.8	40.8
2000	102.0	81.6	61.2	61.2	61.2	61.2	61.2
5000	224.5	163.3	142.9	122.4	122.4	122.4	122.4
10000	217.6	165.8	134.7	124.4	114.0	114.0	114.0
20000	214.1	161.9	130.5	120.1	114.9	109.7	109.7
50000	212.0	159.5	128.0	117.5	111.2	109.1	107.0
100000	211.2	158.7	127.2	116.7	111.4	108.3	107.2

q = 100 bytes

n \ n/i	1	2	5	10	20	50	100
1000	61.2	40.8	40.8	40.8	40.8	40.8	40.8
2000	102.0	81.6	61.2	61.2	61.2	61.2	61.2
5000	114.0	82.9	72.5	62.2	62.2	62.2	62.2
10000	109.7	83.6	67.9	62.7	57.4	57.4	57.4
20000	107.5	81.3	65.5	60.3	57.7	55.0	55.0
50000	106.1	79.9	64.1	58.9	55.7	54.7	53.6
100000	105.7	79.4	63.6	58.4	55.7	54.2	53.6

q = 200 bytes

Figure 7.5 Directory storage ratio R_d for single image per relation and various values of relation cardinality n, ratio of relation cardinality to image cardinality (n/i), and length of a DBC record,q

n \ n/i	1	2	5	10	20	50	100
1000	61.2	40.8	40.8	40.8	40.8	40.8	40.8
2000	51.8	41.5	31.1	31.1	31.1	31.1	31.1
5000	46.0	33.5	29.3	25.1	25.1	25.1	25.1
10000	44.1	33.6	27.3	25.2	23.1	23.1	23.1
20000	43.1	32.6	26.3	24.2	23.1	22.1	22.1
50000	42.5	32.0	25.7	23.6	22.3	21.9	21.5
100000	42.3	31.8	25.5	23.4	22.3	21.7	21.5

q = 500 bytes

n \ n/i	1	2	5	10	20	50	100
1000	31.1	20.7	20.7	20.7	20.7	20.7	20.7
2000	26.1	20.9	15.7	15.7	15.7	15.7	15.7
5000	23.1	16.8	14.7	12.6	12.6	12.6	12.6
10000	22.1	16.8	13.7	12.6	11.6	11.6	11.6
20000	21.6	16.3	13.1	12.1	11.6	11.0	11.0
50000	21.3	16.0	12.8	11.8	11.2	10.9	10.7
100000	21.2	15.9	12.7	11.7	11.2	10.8	10.7

q = 1000 bytes

n \ n/i	1	2	5	10	20	50	100
1000	15.7	10.4	10.4	10.4	10.4	10.4	10.4
2000	13.1	10.5	7.9	7.9	7.9	7.9	7.9
5000	11.6	8.4	7.4	6.3	6.3	6.3	6.3
10000	11.0	8.4	6.8	6.3	5.8	5.8	5.8
20000	10.8	8.2	6.6	6.1	5.8	5.5	5.5
50000	10.6	8.0	6.4	5.9	5.6	5.5	5.4
100000	10.6	7.9	6.4	5.8	5.6	5.4	5.4

q = 2000 bytes

Figure 7.5 (continued) Directory storage ratio R_d

(2) the time to search each eligible MAU for records satisfying a predicate conjunct.

To get a handle at the analysis, we make the following practicable assumptions:

(1) For every MAU accessed by the DBC, we allow for an extra processing time in the structure memory (in order to determine the index terms and thus the MAU numbers). Therefore, a constant factor $K > 1$ will be used to multiply the number of MAU accesses, thereby accounting for query processing time in the structure emmory.

(2) Binary search of the index pages, in a GPC environment, takes a negligible amount of time compared to the time to access each page.

(3) The time to access an index page, the time to access a data page and the time to access an MAU are all equal (equal to the latency time plus rotation time needed to access a disk cylinder).

In the ensuing discussion, we consider the two most important types of queries: single-relation queries and two-relation join queries. The analysis is in the style of [5]. The time to execute a query is determined in terms of the number of accesses to the physical blocks.

7.3.1. Single-Relation Queries

A single-relation query is exemplified by the following SEQUEL query which lists the names and salaries of programmers who earn more than $10,000:

```
SELECT    NAME,SAL
FROM      EMP
WHERE     JOB='PROGRAMMER'
AND       SAL>10000
```

This is an example of a single-relation query with a single predicate conjunct. In the general case, there can be a disjunction of X predicate conjuncts, but then the query may be treated as X queries each with a single predicate conjunct. We, therefore, only restrict ourselves to queries with a single predicate conjunct. Furthermore, the predicates are assumed to be simple predicates (i.e., the predicates are simple comparisons of a field with a value) so that they can be matched with an image. More complicated predicates, such as EMP X.MGR = EMP Y.EMPNO, cannot be matched by an image. Finally, since the consideration of links involves a straightforward extension of the analysis given below, we will only consider images.

The following notations are introduced to simplify the ensuing discussion:

n = relation cardinality;

p = number of predicates in the query;

h = coefficient of CPU time (1/h is the number of tuple comparisons which are considered equivalent in cost to one page access);

i = image cardinality;

K = coefficient of DBC structure memory processing time, the time required to determine index terms (K > 1);

f = # of index page accesses per index search in the GPC environment (for a given storage device and given key length, it is a function of the relation cardinality n and the image cardinality i, but normally has a value lying between 2 and 4);

B_g = average number of tuples (of a relation) per data page (subscript g refers to the GPC environment);

B_d = average number of records per MAU (subscript d refers to the DBC environment); and

j = average number of MAUs spanned by a cluster in the DBC.

The optimizer in System R, has the option to select an access strategy among a variety of choices. The most important of these are listed below. In each case, the execution-time ratio R_t may be determined by computing the ratio of the time T_g required to execute a query in the GPC environment to the time T_d required in the DBC environment.

Case 1. A clustering image is available which matches a predicate with the comparison operator '='. Since the expected number of tuples that satisfy the predicate is n/i, the expected number of data pages to be accessed in the GPC environment is $n/(iB_g)$. Since each of the retrieved tuples must now be compared against the other (p-1) predicates, the total time required in the GPC environment is

$$T_g = n/(iB_g) + (p-1)hn/i + f .$$

T_g may, in actuality, be somewhat less because some of the retrieved tuples may have to be eliminated from further consideration even before all the (p-1) predicates have been compared with them. Furthermore, since the number of tuples retrieved, which is n/i, is expected to be very small, we may even neglect the CPU time required for comparing predicates. Therefore, T_g is simplified to

$$T_g = n/(iB_g) + f \quad .$$

In the DBC environment, whenever the equality predicate matches a clustering image, only one cluster need be searched. Therefore,

$$T_d = jK$$

where the factor K accounts for the structure memory processing time. Finally, the execution-time ratio is

$$R_t = T_g/T_d = n/(iB_g jK) + f/(jK)$$

Case 2. A clustering image is available which matches a predicate whose comparison operator is not '='. Assuming that half the tuples of the relation satisfy the predicate, the expected times are

$$T_g = n/(2B_g) + (p-1)hn/2 + f$$

and

$$T_g = nK(2B_d)$$

Case 3. A non-clustering image is available which matches a predicate whose comparison operator is '='. If this image is used in a GPC environment, then one page access will be required for each of the n/i expected tuples that satisfy the predicate. Without the advantage of secondary clustering information (in the query), the DBC has to access the entire relation. Therefore,

$$T_g = n/i + (p-1)hn/i + f$$

and

$$T_d = nK/B_d$$

Case 4. A non-clustering image is available which matches a predicate whose comparison operator is not '='. If this image is used in the GPC environment, then

$$T_g = n/2 + (p-1)hn/2 + f$$

and

$$T_d = nK/B_d$$

Case 5. A clustering image is used which matches no predicate. In this case, all the tuples must be examined in the GPC environment. Therefore,

$$T_g = n/B_g + phn + f$$

and

$$T_d = nK/B_d$$

Case 6. A non-clustering image is used which matches no predicates. Since, we may justifiably assume that every relation has a clustering image (or clustering link), this choice will actually never have to be made in a GPC environment. In any case, if the choice were, indeed, to be made, then

$$T_g = n + phn + f$$

and

$$T_d = nK/B_d$$

Case 7. Suppose there are $p_e \geq 1$ equality predicates and $p_n \geq 1$ non-equality predicates each of which matches an image, then the $(p_e + p_n)$ images may be searched and a TID list generated for each predicate. These lists may be sorted separately and then intersected to determine the final TID list to be searched. We then get,

$$T_g = (n/(i^{Pe}2^{Pn}) + (p_e + p_n)f$$

where we have neglected the predicate comparison time since the final list of TIDs will be very small; we have also neglected the time to sort the TID lists, which may be appreciable if the lists are actually quite long. Notice that when $p_e \geq 2$, the first term in T_g is likely to be quite small as long as the image cardinalities are moderately large. In such a case, we may write

$$T_g = (p_e + p_n)f .$$

In the DBC environment, we have

$T_d = jK$, if an equality predicate matches a clustering image and

$T_d = nK/B_d$, otherwise.

In Figure 7.6, we have tabulated the values of execution time ratio R_t for each of the seven cases mentioned above. We have used the following figures:

$K = 1.2$

$f = 3$

$p = 2$

$h = 0.0001$

$j = 2$

$B_d/B_g = 50$

B_g is taken from the set {5,20,100,500}

n/i is taken from the set {1,2,5,10,50,100}

n is taken from the set {1000,5000,20000,100000}

$(p_e + p_n)$ is taken from the set {2,3}.

The assumption of $B_d/B_g = 50$ requires a little explanation. An MAU in the DBC is a disk cylinder which normally consists of 20 to 40 tracks. The track size to page size ratio in a conventional system usually varies from 1 to 5. Finally, the size of a DBC record varies from 1 to 2 times the size of the corresponding tuple of a conventional system. Taking these factors into consideration, we have arrived at a reasonable figure of 50 for the ratio B_d/B_g.

n/i \ B_g	5	20	100	500
1	1.33	1.27	1.25	1.25
2	1.42	1.29	1.26	1.25
5	1.67	1.35	1.27	1.25
10	2.08	1.46	1.29	1.26
50	5.42	2.29	1.46	1.29
100	9.58	3.33	1.67	1.33

(i) Case 1

n \ B_g	5	20	100	500
1000	42.94	46.75	67.08	168.75
5000	41.94	42.75	47.08	68.75
20000	41.75	42.00	43.33	50.00
100000	41.70	41.80	42.33	45.00

(ii) Case 2

n \ B_d	250	1000	5000	25000
1000	0.83	3.33	16.67	83.34
5000	0.17	0.67	3.33	16.67
20000	0.04	0.17	0.83	4.17
100000	0.01	0.03	0.17	0.83

(iii) Case 3, n/i = 1

n \ B_d	250	1000	5000	25000
1000	1.04	4.17	20.83	104.17
5000	0.21	0.83	4.17	20.83
20000	0.05	0.21	1.04	5.21
100000	0.01	0.04	0.21	1.04

(iv) Case 3, n/i=2

n \ B_d	250	1000	5000	25000
1000	1.67	6.67	33.34	166.68
5000	0.33	1.33	6.67	33.34
20000	0.08	0.33	1.67	8.33
100000	0.02	0.07	0.33	1.67

(v) Case 3, n/i = 5

Figure 7.6. Execution time ratio R_t for single-relation queries

n \ B_d	250	1000	5000	25000
1000	2.71	10.83	54.17	270.85
5000	0.54	2.17	10.83	54.17
20000	0.14	0.54	2.71	13.54
100000	0.03	0.11	0.54	2.71

(vi) Case 3, n/i = 10

n \ B_d	250	1000	5000	25000
1000	11.04	44.17	220.85	1104.27
5000	2.21	8.83	44.17	220.85
20000	0.55	2.21	11.04	55.21
100000	0.11	0.44	2.21	11.04

(vii) Case 3, n/i = 50

n \ B_d	250	1000	5000	25000
1000	21.46	85.84	429.21	2146.04
5000	4.29	17.17	85.84	429.21
20000	1.07	4.29	21.46	107.30
100000	0.21	0.86	4.29	21.46

(viii) Case 3, n/i = 100

n \ B_d	250	1000	5000	25000
1000	104.80	419.21	2096.04	10480.21
5000	104.30	417.21	2086.04	10430.21
20000	104.21	416.83	2084.17	10420.83
100000	104.18	416.73	2083.67	10418.33

(ix) Case 4

n \ B_g	5	20	100	500
1000	42.33	44.33	55.00	108.33
5000	41.83	42.33	45.00	58.33
20000	41.74	41.96	43.13	48.96
100000	41.71	41.86	42.63	46.46

(x) Case 5

Figure 7.6. (continued) Execution time ratio R_t

n \ B_d	250	1000	5000	25000
1000	209.00	836.00	4180.00	20900.00
5000	208.50	834.00	4170.00	20850.00
20000	208.41	833.63	4168.13	20840.63
100000	208.38	833.53	4167.63	20838.13

(xi) Case 6

n \ B_d	250	1000	5000	25000
1000	1.25	5.00	25.00	125.00
5000	0.25	1.00	5.00	25.00
20000	0.06	0.25	1.25	6.25
100000	0.01	0.05	0.25	1.25

(xii) Case 7, $p_e + p_n = 2$, no clustering image

n \ B_d	250	1000	5000	25000
1000	1.88	7.50	37.50	187.50
5000	0.38	1.50	7.50	37.50
20000	0.09	0.38	1.88	9.38
100000	0.02	0.08	0.38	1.88

(xiii) Case 7, $p_e + p_n = 3$,no clustering image

(xiv) Case 7, $p_e + p_n = 2$; if clustering image matches an equality predicate, then $R_t = 2.50$

(xv) Case 7, $p_e + p_n = 3$; if clustering image matches an equality predicate, then $R_t = 3.75$

Figure 7.6. (continued) Execution time ratio R_t

We observe a number of important facts from the tables in Figure 7.6. Whenever there is an equality predicate matching an image (e.g., Case 1 and Case 3), very few pages need to be searched in the GPC environment, because of the choice of large image cardinalities. Therefore, in these cases the term f dominates the value of T_g. In Case 1, the DBC has to search only one cluster because the equality predicate matches a clustering image. In Case 3, however, the DBC has to content-search the entire relation. So, for very large relations and very large records, the GPC environment is clearly more favorable in Case 3. Similar reasoning holds for Case 7 if the clustering image does not match even one of the equality predicates. In all other cases, the DBC performs one or more orders of magnitude better than a conventional system. In short, the DBC works much better than a conventional system, whenever any one of the following holds:

(1) record size is small, say 50 to 200 bytes,

(2) relation is of small or medium size, say less than 20,000 tuples,

(3) many records (say, greater than 50) are satisfied by an equality predicate, so that many records have to be retrieved by either system,

(4) image cardinality is medium, say $n/i > 100$, which will easily be true for large relations (this observation actually follows from 3), or

(5) a given query does not have any equality predicate that matches an image.

The GPC environment, in contrast, works out as good or better than the DBC only when all the following conditions hold:

(1) the relation is large, say greater than 20,000 tuples,

(2) the records are large, say 500 bytes or larger,

(3) the query has an equality predicate that matches an image, and

(4) the cardinality of the above image is very large, say $n/i \leq 10$.

7.3.2 Queries Involving a Join of Two Relations

While the most common type of SEQUEL query is the single-relation query (because such queries appear both as simple statements and also as embedded queries within compound SEQUEL statements), the second most frequently used query is possibly the one involving a join of two relations. An example of such a query is as follows. It lists the names, salaries and department names of programmers located in Evanston.

```
SELECT      NAME, SAL, DNAME
FROM        EMP,DEPT
WHERE       EMP.JOB = 'PROGRAMMER'
AND         DEPT.LOC = 'EVANSTON'
AND         EMP.DNO = DEPT.DNO
```

The most general form of a join query involves restriction, projection and join. The general query has the form:

> Apply a given restriction (which is a single-relation subquery Q1) to a relation R1, yielding a set of tuples $R1_r$. Apply a possibly different restriction (which is another single-relation subquery Q2) to a relation R2, yeilding $R2_r$. Join $R1_r$ and $R2_r$ and project some fields to derive the final response set.

The optimizer of System R determines an access strategy for such queries, based on the characteristics of the two relation. Four possible methods are shown in [5] but the full details of the optimizer, as well as the justification of the various methods, have not yet been published. Based on a study of the above four methods, we will describe a number of different cases. Once again, images alone will be considered. Consideration of links involves a straightforward extension of the techniques described. Because of the enormous number of ways a join query may be executed, we will often generalize a number of possibilities by describing them in terms of the individual single-relation subqueries Q1 and Q2. In such circumstances, we will formulate the query execution time in terms of the execution times of Q1 and Q2.

The following nomenclature is used in the analysis:

E_1, E_2 = average number of external nodes in a B-tree of relations R1 and R2, respectively ;

G_1, G_2 = the best execution times for separate executions of the subqueries Q1 and Q2 in the GPC environment ;

D_1, D_2 = the best execution times for separate executions of the subqueries Q1 and Q2 in the DBC environment ;

n_1, n_2 = cardinalities of relations R1 and R2, respectively ;

i_1, i_2 = average image cardinality of images on attributes of R1 and R2, respectively ;

f = # of index page accesses for searching a given key in the GPC environment ;

j = average number of MAUs spanned by a cluster in the DBC ;

K = coefficient of DBC structure memory processing time ;

B_g = average number of tuples per data page (in GPC environment) ;

B_d = average number of records per MAU (in DBC environment) ;

T_g = execution time of a join query (in GPC environment);

T_d = execution time of a join query (in DBC environment); and

R_t = execution time ratio T_g/T_d .

Since, the number of secondary storage accesses required for a join query is usually quite large, we will ignore the CPU time required for comparing tuples against predicates in the GPC environment. With this simplification in mind, the various cases are discussed below.

Case 1. If clustering images are available on both the join attributes, if there is an equality predicate in each of Q1 and Q2 and if there is an image on each of the attributes, A1 and A2,that match these equality predicates, then the following algorithm (called the TID algorithm) may be used in the GPC environment:

> Using the image on A1, obtain the TIDs of tuples from R1 which satisfy the equality predicate of R1. Sort them and store the TIDs in a file W1. Do the same with R2, using the image on A2 and yielding a TID file W2. Perform a simultaneous scan over the images on the join attributes of R1 and R2, finding the TID pairs of tuples which match on the join attributes. Check each pair (TID1, TID2) to see if TID1 is present in W1 and TID2 in W2. If they are, the tuples are fetched, joined and projected.

The time to execute this algorithm consists of the time to search the images on A1 and A2, the time to scan entire clustering images (on the join attributes) and the time to retrieve the required tuples from the database. Neglecting the sorting time, the approximate expression for Tg is

$$T_g = 2f + (E_1 + E_2 + 2f) + (n_1/i_1) + (n_2/i_2)$$
$$= 4f + E_1 + E_2 + (n_1/i_1) + (n_2/i_2).$$

In the DBC environment, in the worst case, it may be required to use a CONNECT ON command, thus focing a content-search of entire relations R1 and R2. Therefore, in the worst case

$$T_d = K(n_1 + n_2) \ / \ B_d.$$

However, if n_1/i_1 is small, then it may be faster to retrieve the records satisfying the subquery Q1, sort these records by the join attribute and send a separate command for each unique value of the join attribute, in order to retrieve the records satisfying Q2. This process leads to

$$T_d = Kn_1/B_d + Kj(n_1/i_1).$$

On the other hand, if n_2/i_2 is small, then we may start by retrieving the records satisfying Q2 and then use the unique values of the join attribute of these records to retrieve the records satisfying Q1. We then have

$$T_d = Kn_2/B_d + Kj(n_2/i_2).$$

In Figure 7.7, we have tabulated $R_t = T_g/T_d$ for the worst-case value of T_d. We have used the following values: f=3; $n=n_1=n_2$ taken from the set {1000, 5000, 20000, 100000} $E=E_1=E_2 = n/500$; $i=i_1=i_2$; n/i taken from the set {1, 5, 20, 100} B_d taken from the set {250, 1000, 5000, 25000} and K=1.2. We observe that the DBC performs one or more orders of magnitude better except when record size is very large, say greater than 500 bytes (corresponding to $B_d = 1000$).

Case 2. In case there are clustering images on both the join attributes but no other predicate matches an image, then the following algorithm may be used in the GPC environment:

> Perform a simultaneous scan of the images on the join attributes of the two relations. Advance the R1 scan (retrieving the tuples of R1 at the same time) until the next tuple is found which satisfies Q1. Let the join attribute of this tuple have a value V. Advance the R2 scan and fetch all tuples of R2 that have a value V for the join attribute and satisfy Q2. Repeat until the image scans are completed. (By interchanging R1 with R2 and Q1 with Q2 in the statement of the algorithm, we may get yet another algorithm).

Assuming that there is an equality predicate in Q1, whose attribute has a cardinality i_1, the time required to execute the join query consists of the time to scan completely both the clustering images, the time to access all tuples of R1 and the time to access (n_1/i_1) tuples of R2. Therefore,

$$T_g = (E_1 + f) + (E_2 + f) + n_1/B_g + n_1/i_1$$

The worst case value of T_d, on the other hand, is

$$T_d = k(n_1 + n_2)/B_d$$

If n_1/i_1 is small, we may use the method shown in Case1, to derive an actually better performance.

In Figure 7.8, we have tabulated $R_t = T_g/T_d$ for the worst-case value of T_d. The values used in the calculations are: f=3; k=1.2; $n=n_1=n_2$ taken from the set {1000, 5000, 20000, 100000}; $E=E_1=E_2=n/500$; $i=i_1=i_2$; n/i taken from the set {1, 5, 20, 100} $B_d/B_g = 50$; B_g taken from the set {5, 20, 100, 500}. We notice that the DBC demonstrates a performance that is uniformly better than that of a conventional system.

Case 3. If there is a clustering image on the join attribute of R1, and if there is no clustering image on the join attribute of R2, then a conventional system may first retrieve the restriction of R2 (by executing the single-relation subquery Q2

n \ B_d	250	1000	5000	25000
1000	1.88	7.50	37.50	187.50
5000	0.71	2.83	14.17	70.83
20000	0.49	1.96	9.79	48.96
100000	0.43	1.73	8.63	43.13

(i) n/i = 1

n \ B_d	250	1000	5000	25000
1000	2.71	10.83	54.17	270.83
5000	0.88	3.50	17.50	87.50
20000	0.53	2.13	10.63	53.13
100000	0.44	1.76	8.79	43.96

(ii) n/i = 5

n \ B_d	250	1000	5000	25000
1000	5.83	23.33	116.67	583.33
5000	1.50	6.00	30.00	150.00
20000	0.69	2.75	13.75	68.75
100000	0.47	1.88	9.42	47.08

(iii) n/i = 20

n \ B_d	250	1000	5000	25000
1000	22.50	90.00	450.00	2250.00
5000	4.83	19.33	96.67	483.33
20000	1.52	6.08	30.42	152.08
100000	0.64	2.55	12.75	63.75

(iv) n/i = 100

Figure 7.7. Execution time ratio R_t for a join query (Case 1)

n \ B_g	5	20	100	500
1000	21.98	25.42	43.75	135.42
5000	21.40	23.08	32.08	77.08
20000	21.29	22.65	29.90	66.15
100000	21.26	22.53	29.31	63.23

(i) n/i = 1

n \ B_g	5	20	100	500
1000	22.40	27.08	52.08	177.08
5000	21.48	23.42	33.75	85.42
20000	21.31	22.73	30.31	68.23
100000	21.26	22.55	29.40	63.65

(ii) n/i = 5

n \ B_g	5	20	100	500
1000	23.96	33.33	83.33	333.33
5000	21.79	24.67	40.00	116.67
20000	21.39	23.04	31.88	76.04
100000	21.28	22.61	29.71	65.21

(iii) n/i = 20

n \ B_g	5	20	100	500
1000	32.29	66.67	250.00	1166.67
5000	23.46	31.33	73.33	283.33
20000	21.80	24.71	40.21	117.71
100000	21.36	22.94	31.38	73.54

(iv) n/i = 100

Figure 7.8. Execution time ratio R_t for a join query (Case 2)

in the best possible way), sort these tuples by the join attribute of R2 and then use these values (of the join attribute) for scanning the clustering image of R1. In this case, if there is at least one equality predicate in Q2, then

$$T_g = G_2 + (n_2/i_2) + E_1$$

where we have assumed that, because of the clustering image of R1, only one access is required for each unique value of the join attribute.

The worst-case value of T_d is still

$$T_d = k(n_1 + n_2)/B_d.$$

If n_2/i_2 is small, then, in the DBC environment, it may be faster to first retrieve the records satisfying Q2 in the best possible way, and then use the unique values of the join attribute of these records to fetch the records of R1. In that case,

$$T_d = D_2 + (n_2/i_2)jk.$$

If (n_2/i_2) is very small and also much less than D_2, then we have

$$R_t \simeq (G_2 + E_1)/D_2$$

If (n_2/i_2) is very small but not much less than D_2, then the minimum value of R_t is

$$R_t \simeq (G_2 + E_1)/2D_2$$

Thus, whenever $(n_2/i_2)$is small, the magnitude of R_t is at least of the order of value that can be expected for the single-relation query Q2. For large values of (n_2/i_2), T_g is at least of the same order as T_d. We do not tabulate any values, because G_2 and D_2 are unknown quantities depending on the execution time of single-relation queries, specific cases of which have already been treated in Section 7.3.1

Case 4. This does not involve any novel situation. This case is basically similar to Case 3, and it can be derived from Case 3 by interchanging R1 with R2 and Q1 with Q2. The analysis is also similar.

Case 5. As a final possibility, if there is no clustering image on either of the join attributes, then the following algorithm may be used in the GPC environment as well as in the DBC environment:

> Using the best possible method, retrieve the restriction of R1 and sort the records by their join attributes. Do the same for the restriction of R2. Join the two sets of records.

In the DBC environment, however, the best possible way is the usual way of expressing the commands. For example, if a predicate in Q1 matches a clustering attribute, then the DBC automatically uses its directory to guide its search. Furthermore, this entire algorithm may be expressed by a single DBC command using the CONNECT ON clause.

In this case, we have

$$T_g = G_1 + G_2 \quad \text{and} \quad T_d = D_1 + D_2.$$

Therefore, the execution time ratio in this case is of the same magnitude as would be obtained for single-relation queries.

7.4 Summary

In this section we have compared the performance of two different systems that implement the same relational model of data. We have expressed the performance ratio in terms of various database parameters such as clock length, record size, relation cardinality, image cardinality, etc. Typical values of these parameters have then been inserted into these expressions to obtain some concrete numerical results. The result of these computations is as follows:

(1) The mass memory used by the DBC typically varies from 1 to 2 times that used by a conventional system.

(2) The directory memory required in the DBC environment is typically one or two orders of magnitude less than that required in the GPC environment (by a conventional system).

(3) The execution time required for usual SEQUEL queries is normally one or two orders of magnitude faster when the DBC is used.

8. CONCLUDING REMARKS

This report brings to conclusion the second phase of a study, the goal of which is to design, construct and evaluate a hardware architecture that can support existing database management activities. After initial contemplation [17,18] on the viability of such a goal, it was decided that a large-scale database machine may not actually be out of reach either from the cost standpoint or from the technological standpoint. The most important database activity is decidedly the search of data by their content. To store a very large database on a monolithic associative memory is an enormously costly undertaking. Furthermore, at any given moment, only a small fraction of the database (therefore, the memory) is actually in use. The solution envisioned for this study is a partitioned content-addressable memory (PCAM) where each partition or block is individually content-addressable [9] and only one block is accessible at any given time. This removed one of the cost problems facing the construction of database machines. But the use of PCAM for mass storage gave rise to another problem. In order to execute a user transaction with a reasonably good response time, it should be possible to efficiently identify the blocks relevant to the transaction and thereby avoid an exhaustive search of the entire database store. This necessitated the maintenance of directories in a faster storage which is at least one-hundredth the size of the database storage. To ensure fast response, the directory storage (called structure memory) should also be a PCAM. The answer to such a storage seemed to lie in one of the emerging memory technologies, such as magnetic bubble memory, electron beam addressable memory or charge-coupled devices [8].

The second phase of the study was aimed at demonstrating that the architectural design of the DBC is effective. By effective we mean that the DBC ought to be capable of supporting the most important data models, namely, hierarchical, network and relational, in a manner that it at least outperforms conventional computers and systems. That the DBC clearly outperforms a conventional hierarchical or network database management system has been demonstrated in [1,2]. The current report concludes this work by considering the relational data model and system.

To complete the perspective, we briefly outline the future works envisioned for this study. The next phase consists of bringing the individual components of the DBC down to the logic-design level. The design and testing of all microcodes will be done at the same time. Following this, we plan to construct either a full-scale or a scaled-down prototype of the DBC. The final phase

consists of an evaluation of the DBC by supporting on it an existing database.

In this report, as has been indicated earlier, we have tried to show how well a relational database management system can be supported on the DBC. System R has been considered as a typical relational system. We have shown how user queries written in the data sublanguage called SEQUEL, are represented in the form of DBC commands. Other statements in SEQUEL, for example, those used for updating the database are similarly translated. The clustering information provided in the schema in the form of clustering images and clustering links are appropriately used in clustering the DBC records. It has also been shown how the view mechanism, integrity assertions, triggers and authorization can all be supported on the DBC.

Finally, in this report, we have made a performance analysis in which we have compared the storage requirements and query execution times of a conventional relational system versus a DBC-supported relational system. It has been observed that while the mass memory requirement in the DBC is usually between one and two times the requirement in a conventional system, there is a tremendous saving in directory memory and very large reduction in the execution time of queries when a DBC is being used. Specifically, the usual directory memory requirement and query execution times are likely to be one or more orders of magnitude better than those of a conventional system. The reason for this performance enhancement lies in the very large size of the DBC mass memory blocks, content-addressability of each block and the clustering of DBC records primarily by relation names. Because of very large block sizes, directories are small and every mass memory access allows the DBC to inspect a very large number of records. Because of the content-addressability of each block, the response set of a query is usually the same set of records as returned by the DBC. Therefore, no added CPU time is needed to compare the retrieved records against the predicates that form the query. Clustering of all records belonging to any given relation ensures that any single-relation query, whatever its composition, will require at most as many mass memory accesses as there are blocks occupied by the relation. Further speed gains, which do not show up in the analysis, follow due to the various other functional features of the DBC such as hardware sorting, automatic memory management, and hardware to compute the common set-functions such as average, maximum, minimum and sum.

REFERENCES

[1] Hsiao, D.K., D.S. Kerr and F. Ng, "DBC Software Requirements for Supporting Hierarchical Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-1, April 1977.

[2] Banerjee, J., D.K. Hsiao and D.S. Kerr, "DBC Software Requirements for Supporting Network Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-4, June 1977.

[3] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," Comm. ACM, 13, 6, June 1970, pp. 377-387.

[4] Date, C.J., An Introduction to Database Systems, Second Ed., Addison-Wesley, Reading, Mass., 1977.

[5] Astrahan, M.M., et al, "System R: Relational Approach to Database Management," ACM Trans. Database Systems, 1, 2, June 1976, pp. 97-137.

[6] Stonebraker, M., E. Wong, P. Kreps and G. Held, "The Design and Implementation of INGRES," ACM Trans. Database Systems, 1, 3, Sept. 1976, pp. 189-222.

[7] Baum, R.I., D.K. Hsiao and K. Kannan, "The Architecture of a Database Computer - Part I: Concepts and Capabilities," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1, Sept. 1976.

[8] Hsiao, D.K. and K. Kannan, "The Architecture of a Database Computer - Part II: The Design of Structure Memory and Related Processors," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-2, Oct. 1976.

[9] Hsiao, D.K. and K. Kannan, "The Architecture of a Database Computer - Part III: The Design of the Mass Memory and its Related Components," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-3, Dec. 1976.

[10] Codd, E.F., "Further Normalization of the Data Base Relational Model," in Courant Computer Science Symposium 6: Data Base Systems, Prentice-Hall, Englewood Cliffs, N.J., May 1971, pp. 65-98.

[11] Bernstein, P.A., "Synthesizing Third Normal Form Relations from Functional Dependencies," ACM Trans. Database Systems, 1, 4, Dec. 1976, pp. 277-298.

[12] Fagin, R., "Multivalued Dependencies and a New Normal Form for Relational Databases," ACM Trans. Database Systems, 2, 3, Sept. 1977, pp. 262-278.

[13] Chamberlin,D.D. and R.F. Boyce, "SEQUEL: A Structured English Query Language," Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974, pp. 249-264.

[14] Chamberlin, D.D., et al, "SEQUEL 2: A Unified Approach to Data Definition Manipulation and Control," IBM Rep. No. RJ1798 (#26096), IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., June 1976.

[15] Griffiths, P.P. and B.W. Wade, "An Authorization Mechanism for a Relational Database System," ACM Trans. Database Systems, 1, 3, Sept. 1976, pp. 242-255.

[16] Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[17] Baum, R.I., "The Architecture of a Secure Database Management System," Ph.D. Dissert., The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-75-8, Nov. 1975.

[18] Baum, R.I. and D.K. Hsiao, "Database Computers - A Step Towards Data Utilities," IEEE. Trans. Computers, C-25, 12, Dec. 1976, pp. 1254-1259.

APPENDIX A — NORMAL FORMS OF RELATIONS

We shall start with some definitions. For a relation R consider two attributes A and B. If, at every instant of time, for every value of A, there exists exactly one value of B in the tuples of R, then B is said to be functionally dependent on A, written $A \rightarrow B$. This definition is generalized in the obvious way to functional dependencies involving compound attributes.

An attribute set (compound attribute) X of a relation R is said to be a key of R if every attribute of R is functionally dependent on X and if no subset of X has this property. An attribute that appears in any key of R is called a prime attribute of R. All other attributes are non-prime. The keys of a relation are normally underlined.

If there are three attribute sets X,Y,Z such that $X \subset Y$, $X \rightarrow Z$ and $Y \rightarrow Z$, then Z is said to be partially dependent on Y. If $Y \rightarrow Z$ and there exists no $X \subset Y$ such that $X \rightarrow Z$, then Z is fully dependent on Y.

An attribute A is transitively dependent upon a set of attributes X if there exists a set of attributes Y such that $X \rightarrow Y$, $Y \not\rightarrow X$ and $Y \rightarrow A$, where $A \notin X$, $A \notin Y$ and A,X,Y are taken from the attributes of a single relation R.

A relation R is in first normal form (or 1NF) if every attribute A assumes values from only single-valued domains. That is, relation-valued domains are excluded from relations. The advantage of a 1NF relation over a general relation is that the database can be viewed as a collection of simple tables (instead of 'tables' of tables) so that a small class of operations is applicable to all relations in the database.

The second and third normal forms are introduced to eliminate certain update anomalies. Consider the functional dependencies of a database as shown in Figure A.1. If we create a 1NF relation R(PRODUCT, SUPPLIER, QTY, CITY, STATE), then insertion-deletion anomalies can occur due to the partial dependency of CITY on the key {PRODUCT,SUPPLIER}. Thus, when a supplier is supplying no parts, his city of origin cannot be recorded in the database.

A relation is in 2NF if it is in 1NF and each of its non-prime attributes is fully dependent upon every key. Thus, the database of Figure A.1 can be represented by two 2NF relations:

R1(PRODUCT, SUPPLIER, QTY)

and R2(SUPPLIER, CITY, STATE)

There is still another problem with R2, even though it is in 2NF. Since STATE is transitively dependent on SUPPLIER via CITY, any time the last

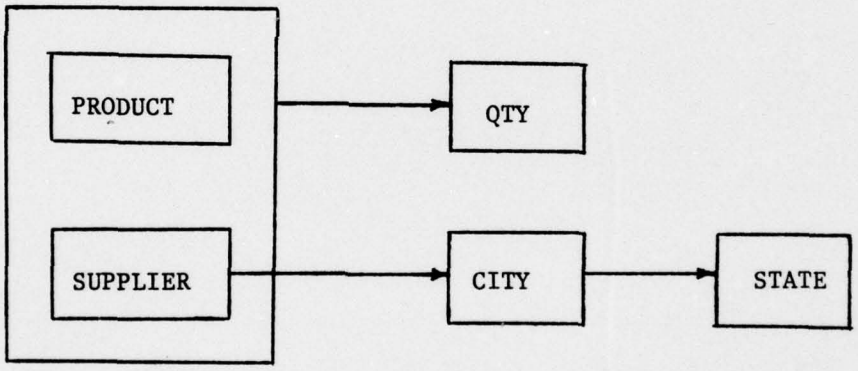
tuple for a city is removed from R2, the corresponding city-state association is simultaneously destroyed.

A relation is in 3NF if none of its non-prime attributes are transitively dependent upon any key. The 2NF relation R2 can be split up into two 3NF relations:

R3(SUPPLIER, CITY)

and R4(CITY, STATE)

Further normalization to a revised version of 3NF relations (called the Boyce-Codd Normal Form) and to 4NF relations (involving multiple dependencies) may be found in [4,12].



PRODUCT, SUPPLIER → QTY

SUPPLIER → CITY

CITY → STATE

Figure A.1. Functional dependencies in a Product-Supplier database.